

The Optimal Pessimistic Transactional Memory Algorithm

Paweł T. Wojciechowski, Konrad Siek
 {pawel.t.wojciechowski,konrad.siek}@cs.put.edu.pl
 Institute of Computing Science, Poznań University of Technology

May 5, 2016

Abstract

Transactional Memory (TM) is an approach aiming to simplify concurrent programming by automating synchronization while maintaining efficiency. TM usually employs the optimistic concurrency control approach, which relies on transactions aborting and restarting if conflicts occur. However, an aborted transaction can still leave some effects in the system that cannot be cleaned up, if irrevocable operations are present within its code. The pessimistic approach eliminates that problem, since it relies on deferring operations in case of conflict rather than aborting, but hitherto pessimistic TMs suffered from low parallelism due to the need of serializing transactions. In this paper, we aim to introduce OptSVA, a pessimistic TM concurrency control algorithm that ensures a high level of parallelism through a battery of far-reaching optimizations including early release, asynchronous execution, and the extensive use of buffering.

Index terms—Transactional memory, concurrency control, pessimistic TM, irrevocable transactions, early release

1 Introduction

In a world dominated by multicore processors and distributed applications even the rank-and-file programmer is increasingly likely to have to turn to parallel programming to take full advantage of various multiprocessor architectures. However, concurrent execution can cause operations on separate processors to interleave in ways that produce anomalous results, forcing the programmer to predict and eliminate them through synchronization.

Yet implementing synchronization correctly is notoriously difficult, since the programmer must reason about interactions among seemingly unrelated parts of the system code. Furthermore, low-level mechanisms like barriers, monitors, and locks are easily misused and performance, consistency, or progress fall prey to faulty design or simple bugs. Worse still, the resulting errors like deadlocks or livelocks are far reaching, difficult to track down, and often non-deterministic.

Consequently, researchers seek ways to automate synchronization while retaining a decent level of efficiency. *Transactional memory (TM)* [12, 20] is one such approach, introduced for multiprocessor architectures, and then extended

to distributed systems as *distributed TM* (see [14, 27], among others). The TM approach requires that the programmer annotates blocks of code as *transactions* that must be executed with specific correctness guarantees (e.g., *serializability* [17], *opacity* [9], and TMS1 [7]). The TM system then ensures these guarantees using an underlying concurrency control algorithm, which provides synchronization as needed to give the illusion of transaction atomicity and isolation, but whose details remain hidden from the programmer. In effect, TM reduces the effort required to implement correct and efficient programs.

Most TM research emphasizes optimistic concurrency control. There are variations to this approach, but, generally, a transaction executes regardless of other transactions using buffers, and only updates the state of the system as it finishes executing (i.e. on *commit*). If two transactions try to access the same object, and one of them writes to it, they *conflict* and one of them *aborts* by discarding its buffers and *restarts* to execute all of its operations anew. Such an approach allows parallelism on multicores, as transactions do not block one another during execution. However, this requires an assumption that an aborted transaction does not have any visible effect on the system.

Hence, the optimistic approach comes with its own set of problems. Most notably, TM transactions can contain any code, including code with side effects, such as: system calls, I/O operations, locking, or network communication. These are referred to collectively as *irrevocable operations*, since it is practically impossible to revoke their effects. However, the *modus operandi* of optimistic transactions depends on aborted transactions cleaning up after themselves. The problem can be mitigated by using irrevocable transactions that run sequentially, and so cannot abort [29], or providing multiple versions of objects on which transactions execute reads [2, 18]. In other cases, irrevocable operations are simply forbidden in transactions (e.g., in Haskell [10]) or moved to commit. Other research suggests that a form of compensation can be used to fix the computations, so that conflicting transactions do not abort [5]. These solutions, however, introduce complexity and overhead, relax the consistency guarantees of TM, or limit the applicability of TM.

A different approach, as suggested in [16, 1, 3] and our earlier work [30, 31], is to use fully-pessimistic concurrency control [4, 28]. This involves transactions waiting until they have permission to access shared objects. In effect, potentially conflicting operations are postponed until they no longer conflict. Thus transactions, for the most part, avoid forced aborts, and therefore, they also naturally avoid the problems stemming from irrevocable operations. However, the authors of [16] show that the fully-pessimistic approach can have negative impact on performance in high contention, since it depends on serializing write transactions to prevent aborts, which inherently limits parallelism. The goal of this paper is to show that this penalty on parallelism is not inherent in the pessimistic approach and can be overcome.

In our previous work [30, 31, 22], we attempted to mitigate the performance issue by introducing a pessimistic TM algorithm using *early release*—a technique that allows conflicting transactions to execute in parallel and still commit. Specifically, the *Supremum Versioning Algorithm (SVA)* uses *a priori* knowledge to allow transactions to safely release a variable it will no longer access, which, in turn, allows other transactions to read or update it without waiting for the first transaction to finish completely. This increases the number of allowed interleavings between transactions, which then translates into

promising performance results.

In this paper we present OptSVA, a TM concurrency control algorithm that builds on the early release mechanism introduced in SVA, but introduces a number of deep modifications that eliminate its predecessors limitations, which effectively make OptSVA a novel and unique algorithm. Most notably, OptSVA parallelizes reads where possible, and relies on buffering to institute automatic *privatization* of variables, where SVA is operation-type-agnostic and writes to variables in-place. This allows to both expedite early release and defer the moment of synchronization, resulting in greater parallelism. Furthermore, to the best of the authors' knowledge, OptSVA is the first TM concurrency control algorithm to delegate specific concurrency-control-related tasks to separate threads to achieve local asynchrony, allowing a transaction to perform local computations and non-conflicting operations while waiting to serialize conflicting operations with other transactions. This feature is especially valuable in distributed systems, where network communication introduces delays.

Furthermore, we show through formal analysis that OptSVA can produce tighter interleavings than SVA due to the increased level of parallelism. This is also born out by experimental evaluation, which verifies that the higher level of complexity of the algorithm does not incur significant enough penalties to nullify the advantage of better interleavings.

In addition, we demonstrate that OptSVA meets the same correctness guarantees as SVA, by presenting a proof for last-use opacity. Last-use opacity [23, 25] is a strong safety property for TM systems, that relaxes opacity to allow early release after the last use of a variable in a transaction.

Given that opacity is defined as a property that must be demonstrated for all prefixes of a given transactional schedule, proving it for a complex system is typically troublesome, as demonstrated by research on markability [15] and graph representation of opacity [9], both techniques trying to work around the basic definition of opacity. By extension, the same is true of last-use opacity. Furthermore, since last-use opacity is defined using histories, but buffering algorithms like OptSVA divorce transactional operations from the actual operations on memory, and perform synchronization based on the latter, demonstrating last-use opacity is even more complex. Hence, apart from the proof itself, we contribute a *trace harmony*, a proof technique that shows last-use opacity based on interrelationships among memory accesses (and can be easily extended to show related properties like opacity).

The paper is structured as follows: Following the introduction, Section 2 shows other research relating to OptSVA. Then, in Section 3 we present the OptSVA algorithm in full. In Section 4 we compare the parallelism of OptSVA histories to those admitted by SVA. In Section ??, we introduce trace harmony, a proof technique that allows us to demonstrate last-use opacity based on memory accesses. Then, in Section 5 we employ that proof technique to show that despite allowing greater parallelism OptSVA satisfies last-use opacity, i.e. the same safety property as SVA. Finally, we conclude in Section 7.

2 Related Work

A large number of TM systems were proposed to date. Here, we concentrate only on those that use some of the same techniques as the one OptSVA is based

on: pessimistic concurrency control and early release.

2.1 Pessimistic TM systems

Seeing as TM systems tend heavily towards optimistic concurrency control, pessimistic systems are relatively rare. Examples of these include our previous work on the Basic Versioning Algorithm and the Supremum Versioning Algorithm [30, 31]. The former is an opaque in-place TM that never aborts transactions and uses *a priori* knowledge on access sets to enforce disjoint-access parallelism. The latter adds an early release mechanism in an effort to allow conflicting transactions to execute partially in parallel. SVA was later extended to also allow optional aborts in [22]. All of these algorithms were proposed for a system model using complex objects defining custom methods rather than variables, so none of them distinguish between reads and writes as in the traditional TM model. The algorithm proposed here builds on both of these and introduces the distinction between operation types, as well as other modifications, all aiming to increase the degree of parallelism the of which TM system is capable.

Another example is the system proposed in [16], where read-only transactions execute in parallel, but transactions that update are synchronized using a global lock to execute one-at-a-time. This idea was improved upon in Pessimistic Lock Elision (PLE) [1], where a number of optimizations were introduced, including encounter-time synchronization, rather than commit-time. However, the authors show that sequential execution of update transactions yields a performance penalty. In contrast, the algorithm proposed in this paper maintains a high level of parallelism regardless of updates. In particular, the entire transaction need not be read-only for a variable that is read-only to be read-optimized.

SemanticTM is another pessimistic TM system [8]. Rather than using versioning or blocking, transactions are scheduled and place their operations in bulk into a producer-consumer queues attached to variables. The instructions are then executed by a pool of non-blocking executor threads that use statically derived access sets and dependencies between operations to ensure the right order of execution. The scheduler ensures that all operations of one transaction are executed before another's. In addition, statically derived access sets and dependencies between operations are used to ensure that operations are executed in the right order. Contrary to SVA and OptSVA, the transactions cannot abort, forcibly, but also do not allow for manual aborts. SemanticTM and versioning algorithms produce similar histories, but while the latter are deadlock-free, SemanticTM is wait-free. However, even without aborts, in contrast to SVA and OptSVA, SemanticTM does not guarantee that a given operation is executed (at most) once.

While not exactly a pessimistic system *per se*, Twilight STM [5] relaxes isolation to allow conflicting transactions to reconcile using so-called twilight code at the end of the transaction and commit nevertheless. If a transaction reads a value that was modified by another transaction since its start, twilight code can re-read the changed variables and re-write the variables the transaction modified to reflect the new state, allowing the transaction to commit anyway. Even though the operations are re-executed, as per optimistic concurrency control, it means that transactions that execute twilight code always finish successfully nevertheless. This means, however, that regardless of transactions aborting or committing, the code within them is prone to re-execution, which introduces

problems with irrevocable operations that SVA and OptSVA try to avoid.

2.2 Early release TM systems

A number of TM systems employ early release to improve parallelism. One example is SVA, which we elaborate on earlier.

Another example is Dynamic STM [11], the system that can be credited with introducing the concept of early release in the TM context. Dynamic STM allows transactions that only perform read operations on particular variables to (manually) release them for use by other transactions. However, it left the assurance of safety to the programmer, and, as the authors state, even linearizability cannot be guaranteed by the system. In contrast, versioning algorithms guarantee, at minimum, last-use opacity.

The authors of [26] expanded on the work above and evaluated the concept of early release with respect to read-only variables on several concurrent data structures. The results showed that this form of early release does not provide a significant advantage in most cases, although there are scenarios where it would be advantageous if it were automated. We use a different approach in SVA and OptSVA, where early release is not limited to read-only variables.

DATM [19] is another noteworthy system with an early release mechanism. DATM is an optimistic multicore-oriented TM based on TL2 [6], augmented with early-release support. It allows a transaction T_i to write to a variable that was accessed by some uncommitted transaction T_j , as long as T_j commits before T_i . DATM also allows transaction T_i to read a speculative value, one written by T_j and accessed by T_i before T_j commits. DATM detects if T_j overwrites the data or aborts, in which case T_i is forced to restart. DATM allows all schedules allowed by conflict-serializability. This means that DATM allows overwriting, as well as cascading aborts. It also means that it does not satisfy last-use opacity. Hence, DATM is weaker than OptSVA (as well as most TM systems). DATM can also incur very high transaction abort rates, in comparison to OptSVA, whose abort rate will tend towards zero (depending on the use of programmatic aborts).

3 OptSVA

This section describes the *Optimized Supremum Versioning Algorithm (OptSVA)*. OptSVA is specified in full in Fig. 1. Given the complexity of the algorithm we split the presentation into four parts. In the first, we explain the rudiments of the use of versioning for concurrency control, as well as the early release mechanism. These are the foundations of the algorithm and the elements which reflect the basic design of SVA. Then, in the other three parts we discuss how the combination of explicit read/write distinction, buffering, and asynchronous execution of specific synchronization-related tasks is used to optimize accesses to read-only variables, to delay synchronization of the initial operation upon an initial write, and expedite early release to the last (closing) write. First, though, we present the system model.

3.1 Transactional Memory System Model

OptSVA operates in a system composed of a set of processes $\Pi = \{p_1, p_2, \dots, p_n\}$ concurrently executing a set of finite sequential programs $\mathbb{P} = \{\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_n\}$, where process p_i executes \mathcal{P}_i . Within these programs there is code that defines transactions. A *transaction* $T_i \in \mathbb{T}$ is some piece of code executed by process p_k , as part of subprogram \mathcal{P}_k . Each process executes transactions sequentially, one at a time. Transactions contain local computations (that can be whatever) and invoke operations on shared variables, or *variables*, for short. Each variable, denoted x, y, z etc. supports the following *operations*, that allow to retrieve or modify its state:

- a) *write* operation (denoted $w_i(x)v$) that sets the state of x to value v ; the operation's *return value* is the constant ok_i , indicating correct execution, or the constant A_i which indicates that transaction T_i has been forcibly aborted due to some inconsistency,
- b) *read* operation $r_i(x)$ whose *return value* is the current state of x , or A_i in case of a forced abort,

In addition, transactions can execute the following operations related to themselves:

- c) *start* (denoted $start_i$) which initializes transaction T_i , and whose return value is the constant ok_i ,
- d) *commit* (denoted $tryC_i$) which attempts to commit T_i and returns either the constant C_i , which signifies a successful commitment of the transaction or the constant A_i in case of a forced abort,
- e) *abort* (denoted $tryA_i$) which aborts T_i and returns A_i .

The operations a–e defined above are part of the so-called transactional API. They can only be invoked within a transaction.

Even though transactions are parts of subprograms evaluated by processes, it is convenient to talk about them as separate and independent entities. Thus, rather than saying p_k executes some operation as part of transaction T_i , we will simply say that T_i executes (or performs) some operation.

Transactions follow a particular life-cycle. At the outset each transaction T_i executes $start_i$, following which it can execute local computations and operations on variables. Eventually, T_i executes either commit or abort, which complete the execution of the transaction, following which the transaction runs no further code. Furthermore, any operation other than start can return A_i , which also means that the transaction was forced to execute abort during that operation. In addition, a transaction can execute abort arbitrarily. In any case, the transaction can perform no other computations following an execution of an abort. If the transaction was forcibly aborted, though, it will be restarted by the system, but it is easier to think of that as a separate consecutive transaction (i.e. $T_j, i \neq j$) executed by the same process.

If transaction T_i executed a read operation on variable x , we say x is in T_i 's *read set*. write operation on x , we say x is in T_i 's *write set*. If T_i executed either a read or a write operation on x , x is in T_i 's *access set*, which we denote $x \in \text{ASet}_i$

```

1 proc start(Transaction  $T_i$ ) {
2   for( $x \in \text{ASet}_i$  sorted by  $<_{lk}$ )
3     lock  $lk(x)$ 
4   for( $x \in \text{ASet}_i$ ) {
5      $gv(x) \leftarrow gv(x) + 1$ 
6      $pv_i(x) \leftarrow gv(x)$ 
7   }
8   for( $x \in \text{ASet}_i$  sorted by  $<_{lk}$ )
9     unlock  $lk(x)$ 
10  for( $x \in \text{ASet}_i$ :  $x$  is read-only)
11    async run read_buffer( $T_i, x$ )
12      when  $pv_i(x) - 1 = lv(x)$ 
13 }

14 proc read(Transaction  $T_i$ , Var  $x$ ) {
15   if ( $x$  is read-only) {
16     join with read_buffer( $T_i, x$ )
17   } else if ( $wc_i(x) = 0$  and  $rc_i(x) = 0$ ) {
18     wait until  $pv_i(x) - 1 = lv(x)$ 
19     checkpoint( $T_i, x$ )
20      $buf_i(x) \leftarrow st_i(x)$ 
21   }
22   if ( $\exists y: rv_i(y) \neq cv(y)$ )
23     abort( $T_i, x$ ) and exit
24    $rc_i(x) \leftarrow rc_i(x) + 1$ 
25   return  $buf_i(x)$ 
26 }

27 proc read_buffer(Transaction  $T_i$ , Var  $x$ ) {
28    $rv_i(x) \leftarrow cv(x)$ 
29    $buf_i(x) \leftarrow x$ 
30   release( $T_i, x$ )
31   async run read_commit( $T_i, x$ )
32     when  $pv_i(x) - 1 = ltv(x)$ 
33 }

34 proc read_commit(Transaction  $T_i$ , Var  $x$ ) {
35    $cv(x) \leftarrow pv_i(x)$ 
36   if ( $\exists y: rv_i(y) > cv(y)$ )
37     abort( $T_i$ ) and exit
38    $ltv(x) \leftarrow pv_i(x)$ 
39 }

40 proc write(Transaction  $T_i$ , Var  $x$ , Value  $v$ ) {
41   if (not  $v$  in domain of  $x$ )
42     abort( $T_i$ ) and exit
43   if ( $\exists y: rv_i(y) \neq cv(y)$ )
44     abort( $T_i$ ) and exit
45    $buf_i(x) \leftarrow v$ 
46    $wc_i(x) \leftarrow wc_i(x) + 1$ 
47   if ( $wc_i(x) = wub_i(x)$ )
48     async run write_buffer( $T_i, x$ )
49       when  $pv_i(x) - 1 = lv(x)$ 
50 }

51 proc write_buffer(Transaction  $T_i$ , Var  $x$ ) {
52   if (not checkpoint made)
53     checkpoint( $T_i, x$ )
54   if ( $\exists y: rv_i(y) \neq cv(y)$ )
55     abort( $T_i$ ) and exit
56    $x \leftarrow buf_i(x)$ 
57   release( $T_i, x$ )
58 }

59 proc commit(Transaction  $T_i$ ) {
60   for( $x \in \text{ASet}_i$ ) {
61     if ( $x$  is read-only)
62       join with read_comit( $T_i, x$ )
63     else {
64       if ( $wc_i(x) = wub_i(x)$ )
65         join with write_buffer( $T_i, x$ )
66       else
67         catch_up( $T_i, x$ )
68         wait until  $pv_i(x) - 1 = ltv(x)$ 
69         if ( $pv_i(x) - 1 = lv(x)$ )
70            $lv(x) \leftarrow pv_i(x)$ 
71         if ( $wc_i(x) + rc_i(x) > 0$ 
72             and  $rv_i(x) = cv(x)$ 
73             and  $pv_i(x) - 1 > lv(x)$ )
74            $cv(x) \leftarrow pv_i(x)$ 
75       }
76     }
77   if ( $\exists y: rv_i(y) > cv(y)$ )
78     abort( $T_i$ ) and exit
79   for( $x \in \text{ASet}_i$ )
80      $ltv(x) \leftarrow pv_i(x)$ 
81 }

82 proc abort(Transaction  $T_i$ ) {
83   for( $x \in \text{ASet}_i$ ) {
84     wait until  $pv_i(x) - 1 = ltv(x)$ 
85     if ( $wc_i(x) > 0$ 
86         and  $pv_i(x) - 1 > lv(x)$ 
87         and  $rv_i(x) = cv(x)$ ) {
88       if ( $wc_i(x) = wub_i(x)$ )
89         join with write_buffer( $T_i, x$ )
90        $x \leftarrow st_i(x)$ 
91        $cv(x) \leftarrow rv_i(x)$ 
92     }
93     if ( $pv_i(x) - 1 = lv(x)$ )
94        $lv(x) \leftarrow pv_i(x)$ 
95      $ltv(x) \leftarrow pv_i(x)$ 
96   }
97 }

98 proc checkpoint(Transaction  $T_i$ , Var  $x$ ) {
99    $st_i(x) \leftarrow x$ 
100   $rv_i(x) \leftarrow cv(x)$ 
101 }

102 proc release(Transaction  $T_i$ , Var  $x$ ) {
103    $cv(x) \leftarrow pv_i(x)$ 
104    $lv(x) \leftarrow pv_i(x)$ 
105 }

106 proc catch_up(Transaction  $T_i$ , Var  $x$ ) {
107   wait until  $pv_i(x) - 1 = lv(x)$ 
108   if (not checkpoint made)
109     checkpoint( $T_i, x$ )
110   if ( $\exists y: rv_i(y) \neq cv(y)$ )
111     abort( $T_i$ ) and exit
112   if ( $wc_i(x) > 0$ )
113      $x \leftarrow buf_i(x)$ 
114 }

```

Figure 1: Full listing of OptSVA.

In OptSVA specifically, executing $start_i$ translates to executing procedure **start**, $tryC_i$ to executing **commit**, $tryA_i$ to **abort**, $w_i(x)v$ to **write**, and $r_i(x)$ to **read**. If a transaction aborts as a result of some operation (returning A_i), **abort** will also have been executed before the operation returns.

3.2 Versioning Concurrency Control

OptSVA uses four version counters to determine whether a given transaction can be allowed to access a particular shared variable, or whether the access should be deferred to avoid conflicts. The intuition behind how these counters work is by analogy to how the teller may manage a queue in a bank: customers who come into the bank retrieve a ticket with a number from a dispenser and wait before approaching the teller until their number is called. Meanwhile the teller increments the number as she finishes serving each consecutive customer. In the analogy, each customer is a transaction, and the teller is some resource, like a shared variable. The number in the customer's hand is his version for that variable, and it is being compared against the number that is currently being served by the teller—the variable's version. The design gets more involved as more variables and more counters are introduced, and we explain it in detail below.

Whenever a transaction T_i starts, it retrieves a *private version* $pv_i(x)$ for every variable x in its access set (lines 2–9). The access set is assumed to be known *a priori*. The values of private versions received by consecutive transactions are generated from a *global version* $gv(x)$, which is initially 0 and is incremented with each starting transaction that has x in its access set. Hence private versions are unique for a given variable and successive for consecutive transactions. The assignment is also guarded by locks so that it is done atomically. In effect, if one transaction T_i has a greater private version for x (or just *version* for x , for short) than another transaction T_j , then all of its private versions are greater than T_j 's.

Given that private versions ascribe a link between transactions and variables, OptSVA then uses them to permit or deny access to variables. Which transaction can access variable x is defined by its *local version* $lv(x)$. Specifically, the local version of a variable x is always equal to the private version of the transaction T_i that most recently finished working on x , i.e. when T_i commits or aborts it sets $lv(x)$ to $pv_i(x)$ (lines 70 and 94, respectively). The transaction that can access x is the next transaction after the one that stopped using x last. That is, the one whose private version for x is one greater than the local version of x . Hence, in order to access x , T_i must wait until the condition $pv_i(x) - 1 = lv(x)$ is met (see, e.g., line 18). We will refer to this condition as the *access condition*.

An example of how this mechanism works is shown in Fig. 2. The diagram depicts a history consisting of operations executed by transactions on a time axis. Every line depicts the operations executed by a particular transaction. The symbol \circ denotes a complete operation execution. The inscriptions above operation executions denote operations executed by the transactions, e.g. $r_i(x) \rightarrow 1$ denotes that a read operation on variable x is executed by transaction T_i and returns 1, and $w_i(x)1$ denotes that a write operation writing 1 to x is executed by T_i , and $tryC_i \rightarrow C_i$ indicates that T_i attempts to commit and succeeds because it returns C_i . On the other hand, the symbol \mathcal{D} denotes an operation

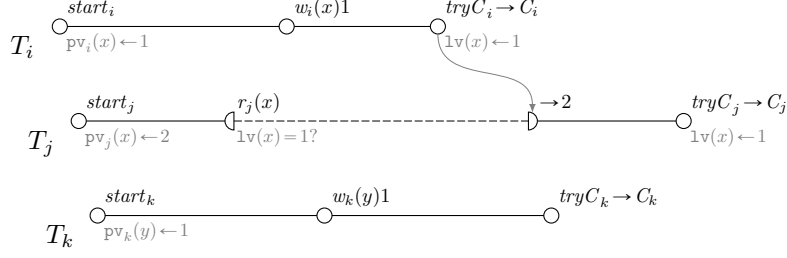


Figure 2: Concurrency control *via* versioning.

execution split into the invocation and the response event to indicate waiting, or that the execution takes a long time. In that case the inscription above is split between the events, e.g., a read operation execution would show $r_i(x)$ above the invocation, and $\rightarrow 1$ over the response. If waiting is involved, the arrow \curvearrowright is used to emphasize a happens before relation between two events. Annotations below events emphasize the state of counters or performed operations within the concurrency control algorithm (used as necessary).

In Fig. 2, T_i and T_j attempt to access shared variable x at the same time. Transaction T_i executes start first, so $pv_i(x) = 1$, and T_j executes second, so $pv_j(x) = 2$. This then determines in which order the transactions access x : initially $lv(x) = 0$, so T_j is not able to pass the access condition and execute its read operation. However, T_i can pass the access condition and it executes its operation without delay. Once T_i commits, it sets $lv(x)$ to 1, so T_j then becomes capable of passing the access condition and finishing executing its read operation. In the mean time, transaction T_k can proceed to access y completely in parallel.

Below we describe the early release mechanism with means to ensure commit order and forced aborts that we use to improve the effectiveness of the version control mechanism.

3.2.1 Early release

The second basic feature of OptSVA is early release based on *a priori* knowledge. Each transaction knows the maximum number of times it will read and write each individual variable at the start of execution. (This information can be provided by the programmer, supplemented by a type checker [30], or generated by static analysis [21].) These *upper bounds* for reads and writes are denoted for transaction T_i and x as, respectively, $\text{rub}_i(x)$ and $\text{wub}_i(x)$. Then, each transaction can count accesses to each variable as they occur using *read* and *write counters*: $\text{rc}_i(x)$ and $\text{wc}_i(x)$ (lines 24 and 46). When the read or write counter for x reaches the upper bound for x , the transaction knows that no further accesses of a particular type will occur afterward. When it is apparent that transaction T_i will perform no further modifications on x , another transaction can start accessing x right away, without waiting for T_i to commit. Hence, if after a write it is true that $\text{wc}_i(x) = \text{wub}_i(x)$ (line 47), then procedure **release** is (eventually) executed and T_i sets $lv(x)$ to $pv_i(x)$.

This is illustrated further in Fig. 3. Here, transaction T_i and transaction T_j both try to access x . Like in Fig. 2, since T_i 's private version for x is lower

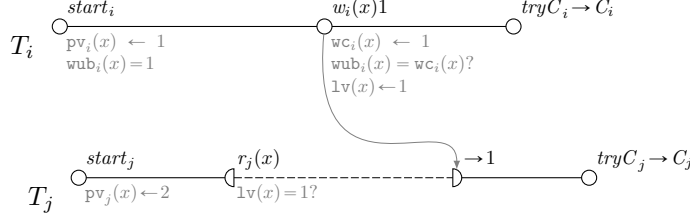


Figure 3: Early release *via* upper bounds.

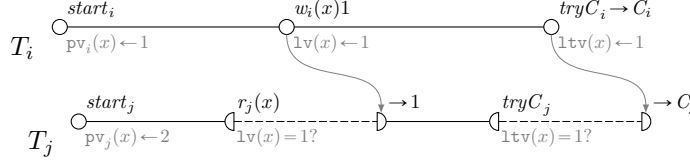


Figure 4: Commit ordering preservation.

than T_j 's, the former manages to access x first, and T_j waits until T_i is released. Unlike in Fig. 2, T_i has upper bound information on writes on x via $\mathbf{wub}_i(x)$: it knows that it will not read from x and will write to x at most once. So, T_i releases x immediately after its write to x , rather than waiting to do so until commit. In effect, T_j can access x earlier.

3.2.2 Commit Order

Although OptSVA is pessimistic and prevents transactions from aborting on conflict, for expressiveness, it allows the programmer to manually invoke the abort operation. Therefore, it is possible for any transaction spontaneously to abort in effect. Such manual aborts can be useful to the programmer to implement conditional rollbacks, and to the system to recover from failures (e.g., in distributed TM). However, this design decision also makes it necessary to enforce the order in which transactions commit to prevent a situation where transaction T_i releases x early and subsequently aborts, but before T_i does abort, T_j reads x and commits. That would mean that T_j committed having acted on an invalid, inconsistent value of x , which is incorrect behavior (e.g., according to serializability [17]).

OptSVA prevents that sort of erroneous situation by ordering commits in the same order as accesses to variables and forcing an abort if the previous transaction also aborted. This is instituted through *local terminal versions*, denoted $\mathbf{1tv}(x)$, that specify which transaction that used x last completed by either committing or aborting. I.e., each transaction writes its private version to $\mathbf{1tv}(x)$, if it only used x , just before finishing the **commit** or **abort** procedure (lines 80 and 95). Then, the local terminal version is used to enforce commit order, as every committing or aborting transaction must wait at *termination condition* $\mathbf{pv}_i(x) - 1 = \mathbf{1tv}(x)$ (lines 68 and 84) before it can commit or abort. Thus, a transaction that accesses x does not complete until the last transaction that previously accessed x commits or aborts.

An example of how this mechanism affects execution is shown in Fig. 4.

Here, T_i , T_j both access x and they respectively get the values of 1 and 2 of \mathbf{pv} for x . Transaction T_i accesses x first and releases it early, setting $\mathbf{lv}(x)$ to 1. This allows T_j to pass the access condition and read x . Transaction T_j subsequently attempts to commit. However, in order to commit T_j must pass the termination condition $\mathbf{pv}_j(x) - 1 = \mathbf{ltv}(x)$, which will not be satisfied until T_i sets $\mathbf{ltv}(x)$ to its own $\mathbf{pv}_i(x)$. Hence T_j can only complete to commit after T_i commits. In general, this condition is checked for every variables in the access set \mathbf{ASet}_j of transaction T_j . In effect the commit order preserves the private version order of transactions for every variable.

3.2.3 Forced Aborts

Furthermore, if some transaction reads a value written by another, and the latter aborts, then the former cannot be allowed to commit having possibly acted upon inconsistent state. Hence, the transaction must be forced to abort.

To enforce aborts, OptSVA marks which version of a variable is the last consistent version via its *current version* $\mathbf{cv}(x)$ —a counter shared by all transactions. This is used in conjunction with its *recovery version* $\mathbf{rv}_i(x)$ —the last consistent version seen by T_i —to check whether transaction T_i is using a consistent (current) version or not. Whenever transaction T_i gains access to shared variable x for the first time, it runs procedure **checkpoint** where it reads the state of x and stores it in its buffer $\mathbf{st}_i(x)$ (line 99). It then sets its recovery version for x $\mathbf{rv}_i(x)$ to x 's current version (line 100). Since $\mathbf{cv}(x)$ is set to some transaction T_j 's private version for x (line 103) whenever T_j releases x or commits (lines 103 and 74), then $\mathbf{rv}_i(x)$ is equal to the private version of a transaction that most recently finished operating on x . Then, whenever some transaction T_j aborts and restores shared variable x from the backup copy (line 90) it also sets $\mathbf{cv}(x)$ back to $\mathbf{rv}_j(x)$ (line 91).

In addition, whenever a transaction tries to commit or access a shared variable, it must test the consistency of all the variables it operates on. Thus, e.g., T_i cannot proceed to access x unless $\mathbf{rv}_i(y) = \mathbf{cv}(y)$ for each variable y in its access set, and must abort otherwise (e.g. lines 22–23). Similarly, if T_i attempts to commit, there must be no variable y in its access set for which $\mathbf{rv}_i(y) > \mathbf{cv}(y)$, or T_i must be forced to abort (line 77–78). Hence, if T_i gains access to x after the previous T_j releases it, and T_j subsequently aborts and sets $\mathbf{cv}(x)$ to a new (lesser) value $\mathbf{rv}_j(x)$, then T_i will be forced to abort either when accessing x later, or when attempting to commit. The condition for aborting is always checked for all variables rather than just the one being accessed, in order to abort as quickly as possible, and to prevent the transaction from operating on both consistent and invalidated variables simultaneously.

We show an example of this in Fig. 5. Here, T_i and T_j access x and have private values for x equal to 1 and 2, respectively. Hence T_i accesses x first. As this is executed T_i sets its recovery version to 0, the value of the current version for x . Then, after the write operation finishes executing, the transaction releases x by setting the local version to 1 and sets the current version to its own private version, i.e. 1. Subsequently T_j meets the access condition and accesses x for the first time, setting its own recovery version to 1 (as $\mathbf{cv}(x) = 1$). Since $\mathbf{rv}_i(x) = \mathbf{cv}(x)$, the access is successful. However, as T_j tries to commit, it is delayed because it cannot satisfy the commit condition. Meanwhile transaction T_i aborts. As it does so, it sets the current version to its recovery version

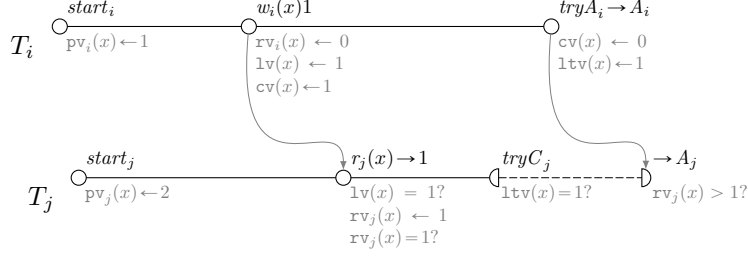


Figure 5: Forced abort.

equal to 0. Then, T_i sets the local terminal version to its own private version, allowing T_j to resume committing. However, T_j cannot satisfy the condition $rv_j(x) > cv(x)$ during commit, since $rv_i(x) = 1$ and $cv(x) = 0$. Hence, T_j is forced to abort.

Note that if no programmatic aborts occur in the system, the system does not experience any forced aborts whatsoever. If the programmer does issue manual aborts, however, cascading aborts can occur.

Below we proceed to describe the three optimizations with respect to the version control mechanism that are employed by OptSVA.

3.3 Read-only Variables

Since originally versioning algorithms did not distinguish between reads and writes, they did not allow read-only transactions to be executed in parallel to other read-only transactions. This is a run-of-the-mill optimization found in all but a small number of TMs, so it is also introduced in OptSVA. However, OptSVA goes a step further, and allows partial parallelization of transactions whenever a variable in a transaction is only read from and not written to, without requiring that all the variables in a transaction are not written to.

Whenever transaction T_i accesses x in such a way that it reads from x but does not write to x (and this is known *a priori*—i.e., $rub_i(x) > 0$ and $wub_i(x) = 0$), we will refer to x as being a *read-only variable* in T_i . In the case of such variables, OptSVA can optimize the accesses by buffering the variable and reading the buffer instead of the actual variable. In addition, since all the reads will be done using the buffer, and the upper bounds indicate that no writes will follow, the variable can be released after it is buffered, irrespective of what operations the transaction will execute later.

Obviously, it is best for parallelism to release any variable as soon as it is no longer needed by a transaction, because it allows other transactions to start acting sooner. Since read-only variables are not needed after they are buffered, they can be released immediately after this happens. The variable must be buffered before or during the first read operation on it is executed, but it could be buffered before that point, even during transaction start. However, in order to buffer a variable, its state must be viewed, so, for the sake of consistency, buffering within versioning concurrency control must be done only after the transaction passes the access condition. Since waiting at the access condition would prevent the transaction from executing operations on other variables or performing local computations, it is best for parallelism for the transaction not

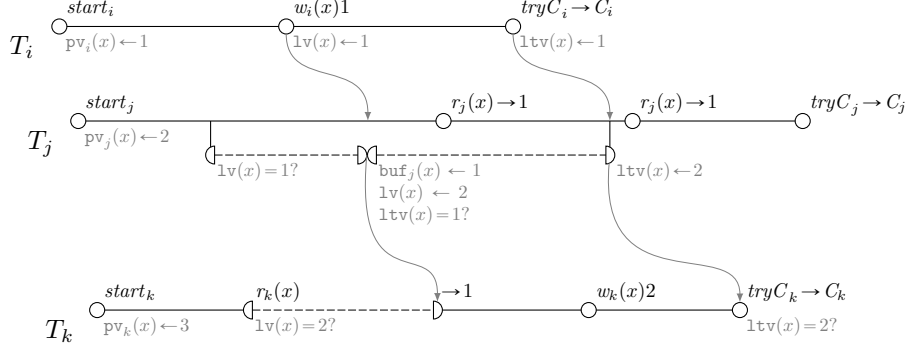


Figure 6: Read-only variable.

to start waiting until it is absolutely necessary.

The algorithm finds balance between buffering as soon as possible and delaying synchronization much as necessary by executing it asynchronously. This is achieved by using the **async run P when C** construct which relegates the execution of procedure **P** to some separate thread. However, before the thread starts executing **P** it waits until condition **C** is satisfied. This allows the transaction to wait at condition **C** without preventing the procedure from delaying other operations that could be executing in the mean time. On the other hand, **P** will be executed as soon as **C** is satisfied, so as soon as it is safe.

OptSVA executes buffering via procedure **read_buffer**. This procedure is relegated to asynchronous execution at lines 11–12, and will execute once the access condition is satisfied. Within **read_buffer**, the transaction T_i saves the value of some variable x to its buffer $\text{buf}_i(x)$ (line 29), and releases it immediately afterward by executing **release** (line 30). Since it is possible that the transaction that wrote the value of x that is being buffered will subsequently abort, T_i also updates its recovery value (line 28), but it does not need to make a checkpoint for x , since the transaction will not modify x . Once read-only variable x is buffered, read operations can use the buffer to retrieve that value, without accessing the variable (line 25), so without waiting. However, a read on a read-only variable cannot be executed until buffering is finished (line 16), which we indicate using the **join with P** construct.

Since a transaction does not modify a read-only variable, if it aborts, it does not need to force other transactions to abort to maintain consistency. Hence, the transaction tries to immediately perform all commit-related operations for a read-only variable immediately after buffering it. This involves waiting for the local terminal version of the object, so by analogy to buffering, the procedure is executed asynchronously, so as not to block other operations. The procedure that executes the commit for variable x is **read_commit** and it is started asynchronously at line 32. The procedure executes a simplified version of **commit** for just x . Hence, once **commit** is executed by the transaction for other variables, it can be skipped for x , and the transaction simply waits for **read_commit** to finish executing.

We show an example of an execution of a transaction T_j with a read-only variable x in Fig. 6. Transaction T_j asynchronously waits for the access condition

on x to be met right after T_j starts, but before any reads actually occur. A parallel line below transaction (such as the one below T_j) indicates procedures executed asynchronously with respect to the thread executing the transaction. Meanwhile T_j can perform local operations or operations on other variables without obstacle. Once T_i releases x , T_j immediately buffers x , and releases it. Then T_j asynchronously tries to commit x , which requires that it waits for the appropriate local terminal version of x . Meanwhile T_k can now access x in parallel to T_j and even write to it, without interfering with T_j 's consistency. Once T_i commits, T_j can then asynchronously commit x , which then allows T_k to commit earlier than it would have otherwise. Since T_j treats x as read-only and hence releases it earlier, transaction T_k is able to execute its operations much sooner, and thus shorten the total execution time of the three transactions.

From the example it is apparent, that the read-only variable optimization moves the point at which such a variable is acquired, released, and committed forward in time. The earlier a shared variable is released by a transaction, the earlier another transaction can start using it, increasing the possibility of acting in parallel, and, therefore, shortening the schedule of execution.

3.4 Delayed Synchronization on First Write

If the first operation that a transaction executes on a particular shared variable is a write operation, then all read operations on that variable are *local*, i.e., they only need to view what the current transaction wrote, and can ignore writes by other transactions. Hence, there is no need for the transaction to synchronize on this variable with other transactions for the sake of those operations. The synchronization is only needed to prevent the current transaction from writing a value to the variable in the middle of another transaction's operations on it. But if the write is saved to a buffer, rather than immediately updating the state of the variable, the synchronization can be delayed until after the write itself, or even after any of the successive read operations.

Since it is beneficial to synchronize as late as possible while performing other tasks beforehand, OptSVA then never checks access conditions on writes (see procedure `write`): either the transaction started with a write, and no synchronization is necessary, or there was a preceding read that already did all the necessary synchronization. Instead, the operation is performed on a buffer (line 45). Then, since all the written values are only visible to the current transaction, the transaction must at some point update the state of the actual variable. This is done either upon executing the last write or during commit. In the former case, when the upper bound on writes is reached (line 47), the transaction asynchronously starts procedure `write_buffer` (line 49), which executes when the access condition is met, and updates the state of the variable (line 56). If the upper bound is not reached during execution, the transaction will instead execute procedure `catch_up` during commit, and update the variable there (line 113), also after waiting at the access condition (line 107).

We illustrate this optimization further in Fig. 7. Here transaction T_i can pass access condition for x first, but nevertheless T_j performs a write simultaneously, since it writes to the buffer rather than wait at the access condition. Transaction T_j only waits at the access condition when it had performed all of its write operations (of which there is one) and starts a separate thread (indicated by the line below) to write the changes to the variable once the access condition

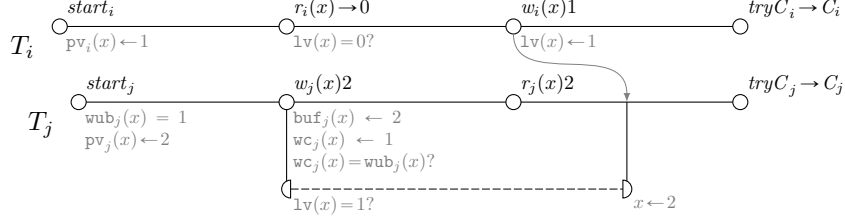


Figure 7: Delayed synchronization on first write.

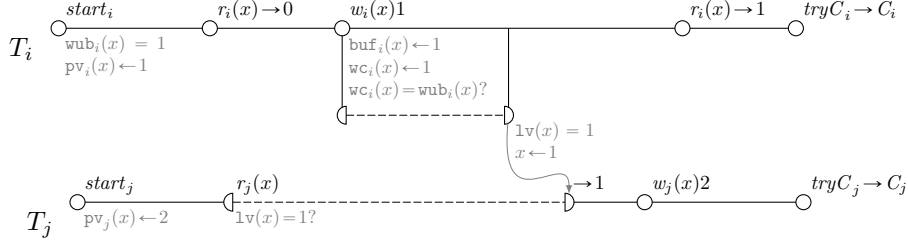


Figure 8: Early release on last write.

is passed. The thread passes the access condition once T_i releases x . Then, T_j applies the value from the buffer to x .

3.5 Early Release on Last Write

Various TMs with early release determine the point at which variables are released variously. For instance, DATM [19] releases variables after each operation, erring on the side of efficiency and guaranteeing only conflict-serializability. SVA, on the other hand, errs on the side of caution and only allows early release after last access to some variable, which it must do because it treats read and write operations uniformly. OptSVA improves on this, since it distinguishes between reads and writes, so early release is done after last write not last access. In effect all reads following last write are executed as if privatized. We argue in [25] that this approach is a solid compromise for TMs with early release.

The early release happens if at some point in the execution of transaction T_i , the upper bound on the number of writes for some variable x is reached when performing a write (line 47). The transaction asynchronously executes **write_buffer** in that instance for the purpose of applying the changes from the buffer to the actual shared variable. After this is done, x will no longer be accessed directly by the transaction, so T_i also executes **release** (at line 57), which sets $lv(x)$ to $pv_i(x)$, which allows other transactions to pass the access condition. Nevertheless, since x was buffered during writes, subsequent reads still have access to a local, consistent value of x (retrieved from the buffer at line 25).

This is illustrated in Fig. 8. Here, T_i knows *a priori* that it will write to x at most once, since $wub_i(x) = 1$. Hence, after the one write to x , a separate thread is started which releases x by setting $lv(x)$ to 1. Since T_i passes the

access condition, this happens almost instantaneously (the figure shows a wait time merely for the reason of aesthetics). Once x is released in this fashion, T_j , whose private version for x is 2, can execute its own read and write operations on x freely. Nevertheless, T_i can continue to execute reads on x after releasing x , and since the value of x is read from T_i 's buffer, T_j 's operations do not interfere.

4 Interleaving Comparison

In this section we compare the interleavings, or *histories*, admitted by OptSVA to those admitted by its predecessor, the Supremum Versioning Algorithm (SVA). SVA (with rollback support) is described in detail in [22, 24]. In short, it amounts to the mechanisms described in Section 3.2, without the optimizations described in Sections 3.3–3.5.

4.1 Preliminaries

In order to compare the interleavings of the two algorithms, let us first provide definitions of transactional histories and relevant ancillary concepts that extend the transactional system model defined in the previous section.

4.1.1 Traces and Operation Executions

Given program \mathbb{P} and a set of processes Π , we denote an execution of \mathbb{P} by Π as $\mathcal{E}(\mathbb{P}, \Pi)$. An execution entails each process $p_k \in \Pi$ evaluating some prefix of subprogram $\mathcal{P}_k \in \mathbb{P}$. The evaluation of each statement $s \in \mathcal{P}_k$ by any process is deterministic. This evaluation produces a (possibly empty) sequence of events (steps) which we denote $\mathbb{L}(s)$.

Furthermore by $\mathbb{L}(\mathcal{P}_k)$ we denote a sequence s.t. given $s_1, s_2, \dots, s_m = \mathcal{P}_k$, $\mathbb{L}(\mathcal{P}_k) = \mathbb{L}(s_1) \cdot \mathbb{L}(s_2) \cdot \dots \cdot \mathbb{L}(s_m)$. By extension, $\mathcal{E}(\mathbb{P}, \Pi)$ produces a sequence of events, which we call a trace \mathbb{T} : $\mathbb{T} \vdash \mathcal{E}(\mathbb{P}, \Pi)$ iff $\forall p_k \in \Pi, \mathcal{P}_k \in \mathbb{P}, \mathbb{L}(\mathcal{P}_k) \subseteq \mathbb{T}$. $\mathcal{E}(\mathbb{P}, \Pi)$ is concurrent, i.e. while the statements in subprogram \mathcal{P}_k are evaluated sequentially by a single process, the evaluation of statements by different processes can be arbitrarily interleaved. Hence, given $\mathbb{T} \vdash \mathcal{E}(\mathbb{P}, \Pi)$ and $\mathbb{T}' \vdash \mathcal{E}(\mathbb{P}, \Pi)$, it is possible that $\mathbb{T} \neq \mathbb{T}'$. We call $\mathcal{E}(\mathbb{P}, \Pi)$ a *complete* execution if each process p_k in Π evaluates all of the statements in \mathcal{P}_k . Otherwise, we call $\mathcal{E}(\mathbb{P}, \Pi)$ a *partial* execution. By extension, if $\mathcal{E}(\mathbb{P}, \Pi)$ is a complete execution, then $\mathbb{T} \vdash \mathcal{E}(\mathbb{P}, \Pi)$ is a *complete* trace.

In order to execute some transactional operation o on variable x within transaction T_i , process p_k issues an *invocation event* denoted $inv_i^k(o)$, and receives a *response event* denoted $res_i^k(u)$, where u is the return value of o . More specifically, given the operations introduced as part of the transactional model, if process p_k executes some operation as part of transaction T_i it issues an invocation event of the form $inv_i^k(start_i)$, $inv_i^k(o)$ for some x , or $inv_i^k(tryC_i)$, (or possibly $inv_i^k(tryA_i)$) and receives a response of the form $res_i^k(u_i)$, where u_i is a value, or the constant ok_i , C_i , or A_i . The superscript always denotes which process executes the operation, and the subscript denotes of which transaction the operation is a part. Each event is atomic and instantaneous, but the execution of the entire operation composed of two events is not.

A pair of these events composed of an invocation of operation o and a response event to o is called a *complete operation execution* and it is denoted

$o_i^k \rightarrow u$, whereas an invocation event $inv_i^k(o)$ without the corresponding response event is called a *pending operation execution*. We refer to complete and pending operation executions as *operation executions*, denoted by op . The transactional model allows the following transactional operation executions (executed by process p_k within transaction T_i as):

- a) $start_i^k \rightarrow ok_i$,
- b) $r_i^k(x) \rightarrow v$ or $r_i^k(x) \rightarrow A_i$,
- c) $w_i^k(x)v \rightarrow ok_i$ or $w_i^k(x)v \rightarrow A_i$,
- d) $tryC_i^k \rightarrow C_i$ or $tryC_i^k \rightarrow A_i$.
- e) $tryA_i^k \rightarrow A_i$.

Since it is convenient to talk about transactions as independent entities, and their relations to specific processes is irrelevant, we will henceforth simplify the notation of invocation and response events to $inv_i(o)$, $res_i(o)$, and of complete executions to $o_i \rightarrow u$. Then, the notation of operation executions becomes $start_i \rightarrow ok_i$, $r_i(x) \rightarrow v$, $w_i(x)v \rightarrow ok_i$, $tryC_i \rightarrow C_i$, etc.

Whenever an operation execution refers to a value, but it is irrelevant to the discussion and inconvenient to specify it, we use a placeholder value \square in its place, writing e.g. $r_i(x) \rightarrow \square$ or $w_i(x)\square \rightarrow ok_i$.

4.1.2 Histories

Given a trace $\mathbb{T} \vdash \mathcal{E}(\mathbb{P}, \Pi)$, a TM *history* H is a subsequence of trace \mathbb{T} consisting only of executions of transactional operations s.t. for every event e , $e \in H$ iff $e \in \mathbb{T}$ and e is either an invocation or a response event specified by the transactional model. If $H \subset \mathbb{T}$ we say \mathbb{T} produces H . A *subhistory* of a history H is a subsequence of H .

The sequence of events in a history H_j can be denoted as $H_j = [e_1, e_2, \dots, e_m]$. For instance, some history H_1 below is a history of a run of some program that executes transactions T_1 and T_2 :

$$H_1 = [inv_1(start_1), res_1(ok_1), inv_2(start_2), res_2(ok_2), \\ inv_1(w_1(x)v), inv_2(r_2(x)), res_1(ok_1), res_2(v), \\ inv_1(tryC_1), res_1(C_1), inv_2(tryC_2), res_2(C_2)].$$

Given any history H , let $H|T_i$ be the longest subhistory of H consisting only of invocations and responses executed by transaction T_i . For example, $H_1|T_2$ is defined as:

$$H_1|T_2 = [inv_2(start_2), res_2(ok_2), inv_2(r_2(x)), res_2(v), \\ inv_2(tryC_2), res_2(C_2)].$$

We say transaction T_i is in H , which we denote $T_i \in H$, if $H|T_i \neq \emptyset$.

Let $H|x$ be the longest subhistory of H consisting only of invocations and responses executed on variable x , but only those that form complete operation executions.

Given complete operation execution op that consists of an invocation event e' and a response event e'' , we say op is in H ($op \in H$) if $e' \in H$ and $e'' \in H$.

Given a pending operation execution op consisting of an invocation e' , we say op is in H ($op \in H$) if $e' \in H$ and there is no other operation execution op' consisting of an invocation event e' and a response event e'' s.t. $op' \in H$.

Given two complete operation executions op' and op'' in some history H , where op' contains the response event res' and op'' contains the invocation event inv'' , we say op' precedes op'' in H if res' precedes inv'' in H .

A history whose all operation executions are complete is a *complete* history.

Most of the time it will be convenient to denote any two adjoining events in a history that represent the invocation and response of a complete execution of an operation as that operation execution, using the syntax $e \rightarrow e'$. Then, an alternative representation of $H_1|T_2$ is denoted as follows:

$$H_1|T_2 = [start_2 \rightarrow ok_2, r_2(x) \rightarrow v, tryC_2 \rightarrow C_2].$$

History H is *well-formed* if, for every transaction T_i in H , $H|T_i$ is an alternating sequence of invocations and responses s.t.,

- a) $H|T_i$ starts with an invocation $inv_i(start_i)$,
- b) no events in $H|T_i$ follow $res_i(C_i)$ or $res_i(A_i)$,
- c) no invocation event in $H|T_i$ follows $inv_i(tryC_i)$ or $inv_i(tryA_i)$,
- d) for any two transactions T_i and T_j s.t., T_i and T_j are executed by the same process p_k , the last event of $H|T_i$ precedes the first event of $H|T_j$ in H or *vice versa*.

In the remainder of the paper we assume that all histories are well-formed.

History H has *unique writes* if, given transactions T_i and T_j (where $i \neq j$ or $i = j$), for any two write operation executions $w_i(x)v' \rightarrow ok_i$ and $w_j(x)v'' \rightarrow ok_j$ it is true that $v' \neq v''$ and neither $v' = v_0$ nor $v'' = v_0$.

4.1.3 Accesses

Given a history H and a transaction T_i in H , we say that T_i *reads* variable x in H if there exists an invocation $inv_i(r_i(x))$ in $H|T_i$. By analogy, we say that T_i *writes* to x in H if there exists an invocation $inv_i(w_i(x)v)$ in $H|T_i$. If T_i reads x or writes to x in H , we say T_i *accesses* x in H . In addition, let T_i 's *read set* be a set that contains every variable x , s.t. T_i reads x . By analogy, T_i 's *write set* contains every x , s.t. T_i writes to x . A transaction's *access set*, denoted $ASet_i$, is the union of its read set and its write set.

Given a history H and a pair of transactions $T_i, T_j \in H$, we say T_i and T_j *conflict* on variable x in H if T_i and T_j are concurrent, both T_i and T_j access x , and one or both of T_i and T_j write to x .

Given a history H and a pair of transactions $T_i, T_j \in H$, we say T_i *reads from* T_j if there is some variable x , for which there is a complete operation execution $w_j(x)v \rightarrow ok_j$ in $H|T_j$ and another complete operation execution $r_i(x) \rightarrow u$ in $H|T_i$, s.t. $v = u$.

Given any transaction T_i in some history H , any operation execution on a variable x within $H|T_i$ is either *local* or *non-local*. Read operation execution $r_i(x) \rightarrow v$ in $H|T_i$ is local if it is preceded in $H|T_i$ by a write operation execution on x , and it is non-local otherwise. Write operation execution $w_i(x)v \rightarrow ok_i$ in $H|T_i$ is local if it is followed in $H|T_i$ by a write operation execution on x , and non-local otherwise.

4.1.4 Execution Time

As program \mathbb{P} is being evaluated by some TM implementation, by a set of processes Π , it takes time to evaluate each statement. Hence, each event e in a trace $\mathbb{T} \vdash \mathcal{E}(\mathbb{P}, \Pi)$ appears at a specific point in time, which we denote $\tau_{\mathbb{T}}(e)$. Since each process p_k executes statements in \mathcal{P}_k in sequence, then, given two events e_1, e_2 s.t. $e_1 <_{\mathbb{T}} e_2$, $\tau_{\mathbb{T}}(e_1) < \tau_{\mathbb{T}}(e_2)$. Given a complete operation execution op consisting of an invocation event e_1 and a response event e_2 , the time at which op finishes executing is $\tau_{\mathbb{T}}^{\rightarrow}(op) = \tau_{\mathbb{T}}(e_2)$. The *execution time* of trace $\mathbb{T} \vdash \mathcal{E}(\mathbb{P}, \Pi)$, denoted $\tau_{\mathbb{T}}$, is equal to the largest execution time for any event in \mathbb{T} . The *release time* of variable x in transaction T_i in \mathbb{T} , denoted $\tau_{\mathbb{T}}^r(T_i)$, is the point in time at which T_i updates $\text{lv}(x)$. The *completion time* of variable x in transaction T_i in \mathbb{T} , $\tau_{\mathbb{T}}^c(T_i)$, is the point in time at which T_i updates $\text{ltv}(x)$.

4.2 Execution Time Comparison

In this section, we show that the execution time of OptSVA histories is lower than than of SVA histories resulting from the execution of the same program by the same processes.

Let $\mathcal{E}_S(\mathbb{P}, \Pi)$ denote a complete execution of program \mathbb{P} by processes Π according to the SVA concurrency control algorithm, and $\mathcal{E}_O(\mathbb{P}, \Pi)$, an otherwise identical execution, but according to OptSVA. Then, there are traces $\mathbb{T}_S \vdash \mathcal{E}_S(\mathbb{P}, \Pi)$ and $\mathbb{T}_O \vdash \mathcal{E}_O(\mathbb{P}, \Pi)$, and histories $H_S = \text{hist}(\mathbb{T}_S)$ and $H_O = \text{hist}(\mathbb{T}_O)$. The histories contain corresponding transactions: if $T_i \in H_S$ then $T_i \in H_O$ and *vice versa*. Let \mathbb{T} be the set of all transactions in H_O and H_S .

For the purpose of the comparison we assume that the events in histories are instantaneous. We also do not account for the time it takes to execute concurrency control code. Finally, we assume that apart from the details of the concurrency control, the execution proceeds the same, regardless of whether it is SVA or OptSVA.

Lemma 1 (Early Release). *For any $T_i \in \mathbb{T}$ and $x \in \text{ASet}_i$, $\tau_{H_O}^c(T_i) \leq \tau_{H_S}^c(T_i)$.*

Proof. An SVA transaction releases x by updating $\text{lv}(x)$ on commit, on abort, and during the last operation execution on x . An OptSVA transactions does so on commit, on abort, during the last write operation execution on x , and after buffering a read-only variable.

- a) If x is a read-only variable an SVA transaction releases x no sooner than the last operation execution on x , so given any read operation execution $r_i(x) \rightarrow \square \in H_S|T_i$:

$$\tau_{H_S}^r(T_i) \geq \tau_{H_S}^{\rightarrow}(r_i(x) \rightarrow \square).$$

On the other hand, OptSVA releases x as soon as possible. That is during $\text{start}_i \rightarrow \text{ok}_i$ at the earliest, and no later than any $r_i(x) \rightarrow v \in H_O|T_i$ at the latest. Thus:

$$\tau_{H_O}^r(T_i) \leq \tau_{H_O}^{\rightarrow}(r_i(x) \rightarrow \square).$$

In that case, all things being equal:

$$\tau_{H_O}^r(T_i) \leq \tau_{H_S}^r(T_i).$$

- b) Alternatively, if the last operation execution in $H_S|T_i|x$ is $r_i(x) \rightarrow \square$, then an SVA transaction releases x no sooner than $r_i(x) \rightarrow \square$, so:

$$\tau_{H_S}^r(T_i) \geq \tau_{H_S}^{\rightarrow}(r_i(x) \rightarrow \square).$$

On the other hand, if last operation execution in $H_O|T_i|x$ is $r_i(x) \rightarrow \square$, then an OptSVA transaction releases x no sooner than any $w_i(x)_{\square} \rightarrow ok_i$ in $H_O|T_i$.

$$\tau_{H_O}^r(T_i) \geq \tau_{H_O}^{\rightarrow}(w_i(x)_{\square} \rightarrow ok_i).$$

Since $\tau_{H_O}^{\rightarrow}(w_i(x)_{\square} \rightarrow ok_i) < \tau_{H_O}^{\rightarrow}(r_i(x) \rightarrow \square)$, then, all things being equal:

$$\tau_{H_O}^r(T_i) \leq \tau_{H_S}^r(T_i).$$

- c) Otherwise, the last operation execution in T_i is $w_i(x)_{\square} \rightarrow ok_i$, so both SVA and OptSVA transactions will release x no sooner than $w_i(x)_{\square} \rightarrow ok_i$, so, all things being equal:

$$\tau_{H_O}^r(T_i) = \tau_{H_S}^r(T_i).$$

□

Lemma 2 (Early Completion). *For any $T_i \in \mathbb{T}$ and $x \in \mathbf{ASet}_i$, $\tau_{H_O}^r(T_i) \leq \tau_{H_S}^r(T_i)$.*

Proof. An SVA transaction updates $\text{ltv}(x)$ on commit, or on abort, so:

$$\tau_{H_S}^c(T_i) = \tau_{H_S}(res_i(C_i)) \text{ or } \tau_{H_S}^c(T_i) = \tau_{H_S}(res_i(A_i)).$$

An OptSVA transactions updates $\text{ltv}(x)$ on commit, on abort, or after releasing a read-only variable. The latter-most potentially precedes a commit, so:

$$\tau_{H_O}^c(T_i) \leq \tau_{H_S}(res_i(C_i)) \text{ or } \tau_{H_O}^c(T_i) \leq \tau_{H_S}(res_i(A_i)).$$

Thus, all things being equal:

$$\tau_{H_O}^c(T_i) \leq \tau_{H_S}^c(T_i).$$

□

Lemma 3 (Early Operation Execution). *For any $T_i \in \mathbb{T}$, and any operation execution op in $H_O|T_i$ and $H_S|T_i$, $\tau_{H_O}^{\rightarrow}(op) \leq \tau_{H_S}^{\rightarrow}(op)$.*

Proof. The case for $\text{pv}_i(x) = 1$ is trivial. If $\text{pv}_i(x) > 1$ then there exists $T_j \in \mathbb{T}$ s.t. $\text{pv}_j(x) + 1 = \text{pv}_i(x)$.

- i) If op is a read operation execution, op can return a value and be a non-local read operation execution, or a local one, or an the operation can return A_i .
- a) If $op = r_i(x) \rightarrow v$ is a non-local read operation execution in both SVA and OptSVA the operation execution will not finish before the access condition is satisfied, so:

$$\tau_{H_S}^{\rightarrow}(op) \geq \tau_{H_S}^r(T_j),$$

$$\tau_{H_O}^{\rightarrow}(op) \geq \tau_{H_O}^r(T_j).$$

Then, from Lemma 1:

$$\tau_{H_O}^r(T_j) \leq \tau_{H_S}^r(T_j),$$

So, all things being equal:

$$\tau_{H_O}^{\rightarrow}(op) \leq \tau_{H_S}^{\rightarrow}(op).$$

- b) If $op = r_i(x) \rightarrow v$ is a local read operation execution, then, by definition, local reads follow a write operation execution, so $\exists op_w = w_i(x) \rightarrow \square \rightarrow \in H_S | T_i$ s.t. $op_w <_{H_S} op$ and $op_w = w_x(v) \rightarrow H_S | T_i$ s.t. $op_w <_{H_S} op$. In that case:

$$\tau_{H_S}^{\rightarrow}(op) \geq \tau_{H_S}^{\rightarrow}(op_w),$$

$$\tau_{H_O}^{\rightarrow}(op) \geq \tau_{H_O}^{\rightarrow}(op_w).$$

Then, from ii:

$$\tau_{H_O}^{\rightarrow}(op_w) \leq \tau_{H_S}^{\rightarrow}(op_w).$$

Hence, all other things being equal:

$$\tau_{H_O}^{\rightarrow}(op) \leq \tau_{H_S}^{\rightarrow}(op).$$

- c) If $op = r_i(x) \rightarrow A_i$, then operation execution in both SVA and OptSVA the operation execution waits until $\text{ltv}(y) = \text{pv}_i(y) - 1$ is true for all $y \in \text{ASet}_i$. This means that each transaction T_k s.t. $\text{pv}_k(y) + 1 = \text{pv}_i(y)$ must update $\text{pv}_i(y)$ to its private version. Hence:

$$\tau_{H_S}^{\rightarrow}(op) \geq \max_{\substack{\forall T_k \in \mathbb{T}, \forall y \in \text{ASet}_k, \\ \text{s.t. } \text{pv}_k(y) + 1 = \text{pv}_i(y)}} \tau_{H_S}^c(T_k),$$

$$\tau_{H_O}^{\rightarrow}(op) \geq \max_{\substack{\forall T_k \in \mathbb{T}, \forall y \in \text{ASet}_k, \\ \text{s.t. } \text{pv}_k(y) + 1 = \text{pv}_i(y)}} \tau_{H_O}^c(T_k).$$

From Lemma 2:

$$\tau_{H_O}^c(T_j) \leq \tau_{H_S}^c(T_j),$$

So, all things being equal:

$$\tau_{H_O}^{\rightarrow}(op) \leq \tau_{H_S}^{\rightarrow}(op).$$

- ii) If op is a write operation execution, op can return ok_i and either be preceded by a non-local read operation execution, or only by write and non-local read operation executions. Otherwise the write operation execution can return A_i .

- a) If $op = w_i(x) \rightarrow \square \rightarrow ok_i$ is preceded by some non-local $op_r = r_i(x) \rightarrow \square$, then in both SVA and OptSVA:

$$\tau_{H_S}^{\rightarrow}(op) \geq \tau_{H_S}^{\rightarrow}(op_r),$$

$$\tau_{H_O}^{\rightarrow}(op) \geq \tau_{H_O}^{\rightarrow}(op_r).$$

From i point a:

$$\tau_{H_O}^{\rightarrow}(op_r) \leq \tau_{H_S}^{\rightarrow}(op_r).$$

Thus, all things being equal:

$$\tau_{H_O}^{\rightarrow}(op) \leq \tau_{H_S}^{\rightarrow}(op).$$

- b) If $op = w_i(x) \square \rightarrow ok_i$ is not preceded by non-local read operation executions, then there is such $op_w = w_i(x) \square \rightarrow ok_i$ (possibly $op_w = op$) such that op_w is the initial operation in $H_S|T_i|x$ and $H_O|T_i|x$. In addition, op_w is necessarily preceded by $op_s = start_i \rightarrow ok_i$, so:

$$\tau_{H_S}^{\rightarrow}(op_w) \geq \tau_{H_S}^{\rightarrow}(op_s),$$

$$\tau_{H_O}^{\rightarrow}(op_w) \geq \tau_{H_O}^{\rightarrow}(op_s).$$

In SVA an initial write waits for the access condition, so:

$$\tau_{H_S}^{\rightarrow}(op) \geq \max(\tau_{H_S}^r(T_j), \tau_{H_S}^{\rightarrow}(op_s)).$$

In OptSVA writes do not wait for the access condition at all, so:

$$\tau_{H_O}^{\rightarrow}(op) \geq \tau_{H_O}^{\rightarrow}(op_s), \text{ regardless of } \tau_{H_S}^r(T_j).$$

From v):

$$\tau_{H_O}^{\rightarrow}(op_s) \leq \tau_{H_S}^{\rightarrow}(op_s).$$

Hence, all other things being equal:

$$\tau_{H_O}^{\rightarrow}(op_s) = \tau_{H_S}^{\rightarrow}(op_s).$$

Then, since either $op_w = op$ or op_w precedes op :

$$\tau_{H_O}^{\rightarrow}(op) \leq \tau_{H_S}^{\rightarrow}(op).$$

- c) If $op = w_i(x) \square \rightarrow A_i$, then, by analogy to ii point c):

$$\tau_{H_O}^{\rightarrow}(op) \leq \tau_{H_S}^{\rightarrow}(op).$$

- iii) If $op = tryC_i \rightarrow \square$, then in both SVA and OptSVA transactions wait until $ltv(y) = pv_i(y) - 1$ is true for all $y \in ASet_i$ before returning from op . This means that each transaction T_k s.t. $pv_k(y) + 1 = pv_i(y)$ must update $pv_i(y)$ to its private version. Hence:

$$\tau_{H_S}^{\rightarrow}(op) \geq \max_{\substack{\forall T_k \in \mathbb{T}, \forall y \in ASet_k, \\ \text{s.t. } pv_k(y) + 1 = pv_i(y)}} \tau_{H_S}^c(T_k),$$

$$\tau_{H_O}^{\rightarrow}(op) \geq \max_{\substack{\forall T_k \in \mathbb{T}, \forall y \in ASet_k, \\ \text{s.t. } pv_k(y) + 1 = pv_i(y)}} \tau_{H_O}^c(T_k).$$

From Lemma 2:

$$\tau_{H_O}^c(T_j) \leq \tau_{H_S}^c(T_j),$$

So, all things being equal:

$$\tau_{H_O}^{\rightarrow}(op) \leq \tau_{H_S}^{\rightarrow}(op).$$

- iv) If $op = tryA_i \rightarrow A_i$, then, by analogy to iv):

$$\tau_{H_O}^{\rightarrow}(op) \leq \tau_{H_S}^{\rightarrow}(op).$$

- v) If $op = start_i \rightarrow ok_i$, then trivially,

$$\tau_{H_O}^{\rightarrow}(op) = \tau_{H_S}^{\rightarrow}(op).$$

□

Corollary 1 (Lower Execution Time). $\tau_{H_O} \leq \tau_{H_S}$.

Thus, the execution time of OptSVA is no worse than SVA. Intuitively, OptSVA is likely perform better in almost all cases though, and especially, if high contention causes many transactions to wait to access the same object—then, the expedited release times and delayed synchronization come into play.

4.3 Practical Comparison [Proposition for Consideration]

Given that the theoretical considerations above ignore the complexity of the concurrency control algorithm itself, it could be argued that the cost of executing individual operations and delegating execution to separate threads are heavy enough to wipe out any theoretical scheduling advantage in practice. Thus, in this section, we perform a practical comparison of the two algorithms, that bears out the conclusions from the previous section, by showing comparing the performance of SVA to OptSVA given a variety of workloads.

For evaluation we used EigenBench [13], a flexible, powerful, and lightweight benchmark that can be used for comprehensive evaluation of mutlicore TM systems by simulating a variety of transactional application characteristics. It generates a traffic of client transactions, which access objects at random (with a specified degree of locality) according to a predefined ratio of reads to writes from three different array types: the hot array contains variables where transactions can conflict, the mild array contains variables accessed transactionally but without the possibility of conflict, and the cold array contains non-transactional variables.

The experiment was run The benchmark was executed on a 10-node cluster with two quad-core Intel Xeon L3260 processors at 2.67 GHz and 4 GB of RAM per node, running OpenSUSE 13.1 (kernel 3.11.10, x86_64 architecture), and connected with a 1Gb network. The implementations of SVA and OptSVA run on the 64-bit Java HotSpot(TM) Java Virtual Machine version 1.8 (build 1.8.0_25-b17), as does the benchmark.

The benchmark executes 80 concurrent threads (one per processor core), each of which executes 10 consecutive transactions. The transactions have three parameters: length, read-to-write ratio, and contention. Long transactions execute 10 operations on shared variables each, while short ones execute 5 each. These operations either have an $5 \div 1$ or $1 \div 5$ read-to-write ratio. The high contention scenarios provide a total of 20 shared variables to the transaction, while the low contention ones provide 80. We only use hot arrays for the purpose of this presentation, since only they impact contention. Locality of operations is at 50% and is based on a 5-variable-long history. We measure total execution time of the entire workload, and throughput—operations executed per second.

The results shown in Fig. 9 confirm the theoretical comparison, showing that the execution time of OptSVA is lower than that of SVA, and that, in practice this is the typical result. The advantage of OptSVA over SVA is affected by contention, since OptSVA optimizations have more impact when the rate of potential conflicts is higher. Thus, in high contention the execution time of OptSVA is 34.7–40.7% lower than that of SVA, whereas in low contention the difference drops to only between 17.7 and 32.9%. Note that the advantage occurs regardless of the fact that the threads executing asynchronous computations for

Parameters	Execution Time [s]		Throughput [ops/s]		Gain [%]
	SVA	OptSVA	SVA	OptSVA	
Short, RW 5 ÷ 1, high cont.	492.2	257.2	8.1	15.5	47.7
Short, RW 1 ÷ 5, high cont.	486.1	266.3	8.2	15.0	45.2
Long, RW 5 ÷ 1, high cont.	994.9	576.0	8.0	13.9	42.1
Long, RW 1 ÷ 5, high cont.	979.7	640.1	8.2	12.5	34.7
Short, RW 5 ÷ 1, low cont.	206.5	169.9	19.3	23.5	17.7
Short, RW 1 ÷ 5, low cont.	210.5	168.8	19.0	23.6	19.8
Long, RW 5 ÷ 1, low cont.	439.9	308.5	18.2	25.9	29.9
Long, RW 1 ÷ 5, low cont.	442.7	297.1	18.1	26.9	32.9

Figure 9: Experimental comparison between SVA and OptSVA.

OptSVA transactions have to share processors with transaction threads. This does not have a large impact, since those threads are mostly waiting at access conditions. In addition, a higher incidence of reads is also better optimized by OptSVA (due to read-only variables), but the difference is not very pronounced (no more than 10%), since the optimization has more impact in long transactions, but read-only variables are increasingly less likely to occur in EigenBench as transactions get longer. On the other hand SVA treats reads and writes the same, so it performs consistently regardless of the read-to-write ratio. The abort rate is 0 in all cases.

5 Correctness

In this section we show that OptSVA satisfies last-use opacity—the same safety property as SVA, meaning that the parallelism optimization does not sacrifice or otherwise relax correctness.

Last-use opacity [23, 25] is a TM correctness property that provides the same guarantees as opacity, with the exception that it allows reading from live transactions after they performed their *closing write*—the last write in that transaction in any possible execution of that program. (For convenience we repeat the definition of the property after the original paper in the appendix.)

Given that OptSVA divorces the operations performed on shared variables within the code of the transaction from the actual accesses to memory that are executed, and since last-use opacity is defined on operations on shared variables, showing correctness is not straightforward. Instead, we use a different method, where we show that the behavior of view and update events in traces generated by OptSVA satisfy a set of specific event-related properties, which we refer to in aggregate as *trace harmony*.

First, we present the preliminary material that defines how operations on memory are represented within traces. Then, we give the definitions making up trace harmony are presented below in this section. We show in Appendix A that any harmonious trace implies a last-use opaque history in general. Finally, we demonstrate that OptSVA traces are harmonious in Section 6, and so, that OptSVA is last-use opaque.

5.1 Events

Events are the results of transactions directly interacting with the memory representing shared variables. When during the execution of some program, some transaction accesses a variable's state (either viewing it or updating it), it issues an update event that is logged in the trace resulting from the execution.

A *view event* $g_i(x)v$ is any event that represents some transaction T_i accessing the state of variable x (i.e. reading the memory location where the value of x is stored) and retrieving the value of v . An *update event* $s_i(x)v$ is any event that represents a modification of the state of variable x by transaction T_i , setting it to the value of v .

Some operations can abort the transaction, rather than doing what they are intended to do. For instance a write operation may fail with an abort rather than setting a new value of some variable. In such cases the transaction will execute specific code that is meant to clean up after the transaction and revert any variables the transaction modified to a previous (consistent) state. We will refer to this code as the recovery procedure. Any update events executed as part of a recovery procedure are called *recovery* (update) events. In contrast, all update events that are not recovery events are called *routine* (update) events. For distinction, we denote a routine update $\circ s_i(x)v$ and a recovery update $\blacklozenge s_i(x)v$.

Given a view event $g_i(x)v$ (for some T_i), v is specified by the most recent preceding update event on x in a given trace. I.e., if the most recent preceding update event on x is some $s_j(x)v'$ (for some T_j), then $v = v'$. Note, that this distinction does not depend on how the events appear in the trace, but is intrinsic to the code that executes them.

Event $e = s_i(x)v$ is the *ultimate update* event on x in \mathbb{T} iff there is no $e' = s_j(x)v'$ s.t. $e <_{\mathbb{T}} e'$. Event $e = s_i(x)v$ is the *ultimate routine update* event on x in \mathbb{T} iff e is routine and there is no $e' = s_j(x)v'$ s.t. $e <_{\mathbb{T}} e'$ and e' is routine.

Given a view event $e_v = g_i(x)\square$ in some T_i and an update event $e = s_j(x)\square$ in some T_j , e prefaces e_v in trace \mathbb{T} , denoted $e \prec_{\mathbb{T}} e_v$ iff $e <_{\mathbb{T}} e_v$ and there is no update event $e' = s_k(x)\square$ in any T_k s.t. $e <_{\mathbb{T}} e' <_{\mathbb{T}} e_v$. Given a read operation execution $op_r \in \mathbb{T}$ s.t., $op_r = r_i(x) \rightarrow v$ and op_r that consists of an invocation event e_i and a response event e_r , and a view event $e_v = g_i(x)v'$, op_r depends on e_v (denoted $op_r \leftarrow e_v$) iff $v' = v$ and $e_v \prec_{\mathbb{T}} e_r$. Given a write operation execution $op_w \in \mathbb{T}$ s.t., $op_w = w_i(x)v \rightarrow ok_i$ and op_w consists of an invocation event e_i and a response event e_r , and an update event $e_u = g_i(x)v'$, op_w instigates e_u (denoted $op_w \rightsquigarrow e_u$) iff $v' = v$ and $e_i <_{\mathbb{T}} e_u$.

Transaction T_i views transaction T_j ($T_i \rightsquigarrow T_j$) if $\exists e_u, e_v \in \mathbb{T}$ s.t. $e_v = g_i(x)v$ and $e_u = \circ s_j(x)v$ and $e_u \prec_{\mathbb{T}} e_v$. Transaction T_i virtually views transaction T_j ($T_i \rightsquigarrow^v T_j$) if $\exists e_u, e_v \in \mathbb{T}$ s.t. $e_v = g_i(x)v$ and $e_u = \circ s_j(x)v$ and $e_u <_{\mathbb{T}} e_v$.

Event access set $ESet_i$ for some transaction $T_i \in \mathbb{T}$ is such a set of variables such that $x \in ESet_i \iff \exists e \in \mathbb{T}|T_i$ s.t. $e = \circ s_i(x)v$ or $e = g_i(x)v$.

Given $T_i \in \mathbb{T}$, s.t. $e_v = g_i(x)v \in \mathbb{T}|T_i$ and e_v is initial in $\mathbb{T}|T_i$, let $\psi_{\mathbb{T}}(T_i, x)$ be such longest sequence of transactions that: a) if $\exists T_j \in \mathbb{T}$ s.t. $e_a = \blacklozenge s_j(x)v \mathbb{T}|T_j$ and $e_a <_{\mathbb{T}} e_v$ then $\psi_{\mathbb{T}}(T_i, x) = \psi_{\mathbb{T}}(T_i, x) \cdot T_j$, otherwise b) $\psi_{\mathbb{T}}(T_i, x) = \emptyset \cdot T_i$.

Let a view chain $\xi(\mathbb{T}, T_i, T_j)$ be a sequence of transactions s.t. T_i is the first element, and T_j is the last element, and for each pair of consecutive transactions T_k, T_l , it is true that $T_l \rightsquigarrow T_k$. Let $H|\xi(\mathbb{T}, T_i, T_j)$ be the longest subsequence of \mathbb{T} s.t. $e \in H|\xi(\mathbb{T}, T_i, T_j)$ iff $e \in \mathbb{T}|T_i$ and $T_i \in \xi(\mathbb{T}, T_i, T_j)$.

5.2 Definitions

Since OptSVA limits events within a transaction to at most a single routine update event, at most a single single recovery update event, and at most a single view event per variable, we limit the method presented below to such a case. This is represented by the definition of *minimalism* below. (However, the method can be extended to allow multiple routine update events and multiple view events per transaction.)

Definition 1 (Minimalism). *Given transaction $T_i \in \mathbb{T}$, for each x , $\mathbb{T}|T_i$ contains:*

- a) *either none or one view event $g_i(x) \square$,*
- b) *either none or one routine update event $\circ s_i(x) \square$,*
- c) *either none or one recovery update event $\blacklozenge s_i(x) \square$.*

Trace isolation stipulates, that once a transaction starts accessing the memory of some variable, it has exclusive access to it until it is done performing routine updates and view events on it. Hence a transaction is not interfered with by other transaction when it is performing memory accesses, unless an abort is required. Furthermore, if one transaction accesses the memory of one variable before another transaction, then that other transaction cannot access any other variable before the first transaction does.

Definition 2 (Trace Isolation). *Trace \mathbb{T} is isolated, iff given any two transactions T_i and T_j in \mathbb{T} for every $x \in ESet_i \cap ESet_j$, it is true that given any event e_i s.t. $e_i = g_i(x)v \in \mathbb{T}|T_i$ or $e_i = \circ s_i(x)v' \in \mathbb{T}|T_j$, and any event e_j s.t. $e_j = g_j(x)v' \in \mathbb{T}|T_j$ or a routine update event $e_j = \circ s_j(x)v' \in \mathbb{T}|T_j$, $e_i <_{\mathbb{T}} e_j$.*

Isolation order imposes an order on transactions in a trace that respects the order of executing update and view events on variables. Given an isolated trace, there exist the following orders:

Definition 3 (Variable Isolation Order). *Two transactions T_i and T_j are isolation-ordered in trace \mathbb{T} with respect to x , which we denote $T_i \dot{<}^x_{\mathbb{T}} T_j$, if given any event e_i s.t. $e_i = g_i(x)v \in \mathbb{T}|T_i$ or $e_i = \circ s_i(x)v' \in \mathbb{T}|T_j$, and any event e_j s.t. $e_j = g_j(x)v' \in \mathbb{T}|T_j$ or a routine update event $e_j = \circ s_j(x)v' \in \mathbb{T}|T_j$, and $e_i <_{\mathbb{T}} e_j$.*

Definition 4 (Direct Isolation Order). *Two transactions T_i and T_j are directly isolation-ordered $T_i \ddot{<}_{\mathbb{T}} T_j$ if for every $x \in ESet_i \cap ESet_j$, $T_i \dot{<}^x_{\mathbb{T}} T_j$.*

Definition 5 (Isolation Order). *Two transactions T_i and T_j are isolation-ordered $T_i \dot{<}_{\mathbb{T}} T_j$ there exists a sequence of transactions $\epsilon = T_i \cdot \dots \cdot T_j$, where for every pair of consecutive transactions $T_n, T_m \in \epsilon$, $T_n \ddot{<}_{\mathbb{T}} T_m$.*

Note that if $T_i <_{\mathbb{T}} T_j$ and $x \in ESet_i \cap ESet_j$, then $T_i \dot{<}^x_{\mathbb{T}} T_j$, so the isolation order preserves real-time order.

Consonance describes when a particular event or operation involve a value that can be considered correct, which is determined by other events or operations that either precede or follow the one in question.

Specifically, a view event is consonant if it retrieves the value that was written there by a preceding event, or the initial value, if no events preceded. A

consonant read operation must then return a value that was retrieved by a view event beforehand. On the other hand, a routine update event must be caused by some write operation. Whereas a consonant recovery update event is one that cleans up after a routine update and returns the state of a variable to a value that was retrieved by a view event that view the unmodified state of the variable in question.

Definition 6 (View Consonance). *Given some $T_i \in \mathbb{T}$, a view event $e_v = g_i(x)v$ is consonant in \mathbb{T} iff either:*

- a) $v = 0$ and $\nexists e_u \in \mathbb{T}$ s.t. $e_u = s_j(x)v'$ for any T_j , and $e_u <_{\mathbb{T}} e_r$,
- b) $v \neq 0$ and $\exists e_u \in \mathbb{T}$ s.t. $e_u = \circ s_j(x)v$ for some T_j , $i \neq j$, $e_u <_{\mathbb{T}} e_r$, and e_u is the ultimate routine update on x in $\mathbb{T}|T_j$, or
- c) $\exists e_u \in \mathbb{T}$ s.t. $e_u = \blacklozenge s_j(x)v$ for some tr_j , $i \neq j$, $e_u <_{\mathbb{T}} e_r$.

Definition 7 (Routine Update Consonance). *Given some $T_i \in \mathbb{T}$, a routine update event $e_u = \circ s_i(x)v$ is consonant in \mathbb{T} iff e_u is instigated in \mathbb{T} by a consonant write operation execution.*

Definition 8 (Recovery Update Consonance). *Given some $T_i \in \mathbb{T}$, event $e_s = \blacklozenge s_i(x)v$ is consonant in \mathbb{T} iff:*

- a) e_a is conservative in \mathbb{T} , i.e. there exists a consonant non-local view event e_v in $\mathbb{T}|T_i$ that is initial in $\mathbb{T}|T_i$,
- b) e_a is needed in \mathbb{T} , i.e. $\exists e_u = \circ s_i(x)v' \in \mathbb{T}|T_i$ s.t. $e_u <_{\mathbb{T}|T_i} e_a$,
- c) e_a is dooming in \mathbb{T} , i.e. $\nexists r \in \mathbb{T}$ s.t. $r = res_i(C_i)$, $e_a <_{\mathbb{T}|T_i} r$,
- d) e_a is ending in \mathbb{T} , i.e. $\nexists e \in \mathbb{T}$ s.t. $e = g_i(x)v'$ or $e = s_i(x)v'$, $e_a <_{\mathbb{T}|T_i} e$,
- e) e_a is clean in \mathbb{T} , i.e. given view e_v that justifies that e_s is conservative, there is no event $e'_a = \blacklozenge s_j(x)v'$ in any T_j s.t. $T_j <_{\mathbb{T}}^x T_i$ and $e_v <_{\mathbb{T}} e'_a <_{\mathbb{T}} e_a$.

Definition 9 (Non-local Read Consonance). *A non-local read operation execution is consonant in trace \mathbb{T} iff it depends in \mathbb{T} on a consonant non-local view event.*

Definition 10 (Local Read Consonance). *Given some $T_i \in \mathbb{T}$, a local read operation execution $op_r = r_i(x) \rightarrow v$ is consonant in trace \mathbb{T} iff there exists $op_w = w_i(x)v \rightarrow ok_i \in \mathbb{T}|T_i$ s.t. $op_w <_{\mathbb{T}|T_i} op_r$, and op_w is consonant.*

Definition 11 (Write Consonance). *A write operation execution $w_i(x)v \rightarrow ok_i$ in some T_i is consonant in trace \mathbb{T} iff $v \neq 0$ and v is within the domain of x .*

Definition 12 (Trace Consonance). *Trace \mathbb{T} is consonant iff all operation executions, update events, and view events in trace \mathbb{T} are consonant.*

Obbligato ensures that update events required by write operations happen on time, so that the values written to variables by operation executions are actually set in memory by the time the transaction relinquishes control of each variable. This means that a routine update event is required after a write operation by the time a transaction commits (*committed write obbligato*), one is required after a closing write operation, before any other transaction attempts

to access that variable (*closing write obligato*), and one is required if a non-aborted transaction executed write operations and another transaction accesses the variables in question (*view write obligato*).

Definition 13 (Committed Write Obligato). *Given $T_i \in \mathbb{T}$, if $\exists op_w \in \mathbb{T}|T_i$ s.t. $op_w = w_i(x)v \rightarrow ok_i$, op_w is non-local, and $\exists r \in \mathbb{T}|T_i$ s.t. $r = res_i(C_i) \in \mathbb{T}|T_i$, then op_w is in obligato iff $\exists e_s \in \mathbb{T}|T_i$ s.t. $e_s = \circ s_i(x)v$ and $op_w \rightsquigarrow e_s$ and $e_s <_{\mathbb{T}|T_i} r$.*

Definition 14 (Closing Write Obligato). *Given $T_i \in \mathbb{T}$, if $\exists op_w \in \mathbb{T}|T_i$ if $\exists T_j \in \mathbb{T}$ s.t. $T_i <_{\mathbb{T}} T_j$ if there is $op_i = w_i(x)\square \rightarrow ok_i \in \mathbb{T}|T_i$, op_i is a closing write, and there is $e_v = g_j(x)\square \in \mathbb{T}|T_j$, then op_i is in closing obligato iff $\exists e_u \in \mathbb{T}|T_i$ s.t. $e_u = \circ s_i(x)v$ and $op_i \rightsquigarrow e_u$, and $e_u <_{\mathbb{T}} e$.*

Definition 15 (View Write Obligato). *Given $T_i \in \mathbb{T}$, if $\exists T_j \in \mathbb{T}$, s.t. $T_i <_{\mathbb{T}} T_j$, if there is $op_i = w_i(x)\square \rightarrow ok_i \in \mathbb{T}|T_i$, and $e_v = g_j(x)\square \in \mathbb{T}|T_j$, then op_i is in view write obligato iff there is $e_u = s_i(x)\square \in \mathbb{T}|T_i$ s.t. $e_u <_{\mathbb{T}} e_v$ or $\exists r = res_i(A_i) \in \mathbb{T}|T_i$ s.t. $r <_{\mathbb{T}} e_v$.*

Definition 16 (Obligato). *Trace \mathbb{T} is obligato iff*

- a) *all non-local writes in all transactions committed in \mathbb{T} are in committed obligato,*
- b) *all closing writes whose effects are potentially viewed are in closing write obligato,*
- c) *all writes whose effects are potentially viewed are in view write obligato.*

Decisiveness is achieved, when transactions do not let other transactions to view the values they set to the variables they modify until they commit or perform their closing writes.

Definition 17 (Decisiveness). *Trace \mathbb{T} is decisive iff given any pair of transactions $T_i, T_j \in \mathbb{T}$, s.t. $T_i \rightsquigarrow T_j$ for any $e_u = \circ s_j(x)v \in \mathbb{T}|T_j$ and $e_v = g_i(x)v \in \mathbb{T}|T_i$, then either T_j is decided on x , $\exists r = res_j(C_j) \in \mathbb{T}|T_j$ s.t. $e_u <_{\mathbb{T}} r <_{\mathbb{T}} e_v$.*

Abort accord is a relation between two transactions, where if one of them views the update events performed by the other, and the other transaction aborts, then the first transaction is not permitted to abort.

Definition 18 (Abort Accord). *Trace \mathbb{T} is in abort accord iff for any two transactions T_i and T_j in \mathbb{T} s.t. a) $T_j \rightsquigarrow T_i$, if T_i is aborted in \mathbb{T} , then T_j is either live or aborted in \mathbb{T} , b) $\exists e_u = \circ s_i(x)\square \in \mathbb{T}|T_i$ and $e = \circ s_j(x)\square \in \mathbb{T}|T_j$ or $e = g_j(x)\square \in \mathbb{T}|T_j$ and $e_a = \blacklozenge s_i(x)\square \in \mathbb{T}|T_i$, and $e_u <_{\mathbb{T}} e <_{\mathbb{T}} e_a$, then T_j is either live or aborted in \mathbb{T} .*

Commit accord is a similar relation, where given two transactions such that one of them views the update events performed by the other, and the first transaction commits, then the first transaction must have also committed.

Definition 19 (Commit Accord). *trace \mathbb{T} is in commit accord iff for any two transactions T_i and T_j in \mathbb{T} s.t. $T_j \rightsquigarrow T_i$, if T_j is committed in \mathbb{T} , then T_i is committed in \mathbb{T} .*

Coherence specifies, that if a transaction commits, all preceding transactions according to the isolation order either committed or aborted beforehand.

Definition 20 (Coherence). *Trace \mathbb{T} is coherent iff for any two transactions T_i and T_j in \mathbb{T} s.t. $T_i \dot{<}_{\mathbb{T}}^x T_j$, if $\exists r_j = \text{res}_j(C_j) \in \mathbb{T}|T_j$, then $\exists r_i = \text{res}_i(C_i)$ or $r_i = \text{res}_i(A_i)$ and $r_i <_{\mathbb{T}} r_j$.*

Abort Coda specifies when a recovery event can be expected to be issued. If a transaction updates the state of some variable and eventually aborts, either it or another transaction will issue a recovery event to clean up that update before the transaction in question completes aborting. On the other hand, if the transaction commits, neither it or any other transaction will issue a recovery event to revert the state of that variable to another value.

Definition 21 (Abort Coda). *Trace \mathbb{T} has coda iff for any transaction T_i*

- a) *if T_i aborts in \mathbb{T} (so $r = \text{res}_i(A_i) \in \mathbb{T}|T_i$), then if $\exists e_u = \circ s_i(x)v \in \mathbb{T}|T_i$, then for some T_j s.t. $i = j$ or $T_i \dot{<}_{\mathbb{T}}^x T_j$ $\exists e_a = \blacklozenge s_i(x)v' \in \mathbb{T}$ s.t. $e_u <_{\mathbb{T}} e_a <_{\mathbb{T}}$,*
- b) *if T_i commits in \mathbb{T} (so $\exists r = \text{res}_i(C_i) \in \mathbb{T}|T_i$), then if $\exists e = \circ s_i(x)v \in \mathbb{T}|T_i$ or $e = g_i(x)v \in \mathbb{T}|T_i$, then for any T_j s.t. $i = j$ or $T_i \dot{<}_{\mathbb{T}}^x T_j$ $\nexists e_a = \blacklozenge s_i(x)v' \in \mathbb{T}$ s.t. $e <_{\mathbb{T}} e_a <_{\mathbb{T}}$.*

Chain consistency describes what events are allowed and barred from a chain of transactions. Specifically, *chain isolation* stipulates that, a chain of transactions executing view and update events is not broken by a revert event, so a transaction cannot view an inconsistent state where the value of one variable is retrieved before an abort was performed, and another one after. *Chain self-containment* if the values viewed by a transaction in some chain always come from that chain.

Definition 22 (Chain Isolation). *Given trace \mathbb{T} , transactions $T_i, T_j \in \mathbb{T}$, $\xi(\mathbb{T}, T_i, T_j)$ is isolated if for $\forall T_k \in \xi(\mathbb{T}, T_i, T_j)$ s.t. $e^k = \circ s_k(x)v$, there is no T_l (possibly $T_l \notin \xi(\mathbb{T}, T_i, T_j)$) s.t. $\exists e^l = \blacklozenge s_l(x)v'$ where $v = v'$ and e^l is between e^k and any other event in any transaction in $\xi(\mathbb{T}, T_i, T_j)$.*

Definition 23 (Chain Self-containment). *Given trace \mathbb{T} , transactions $T_i, T_j \in \mathbb{T}$, $\xi(\mathbb{T}, T_i, T_j)$ is self-contained iff given any transactions $T_k, T_l \in \xi(\mathbb{T}, T_i, T_j)$, s.t. $k \neq l$ and $\exists e_u^k = \circ s_k(x)v \in \mathbb{T}|T_k$ $\exists e_v^l = g_l(x)v' \in \mathbb{T}|T_l$ and $e_u^k <_{\mathbb{T}} e_v^l$, then either $v = v'$ or $\exists e_u^m = \circ s_m(x)v' \in \mathbb{T}|T_m$ for some $T_m \in \xi(\mathbb{T}, T_i, T_j)$ s.t. T_m precedes T_l and follows T_k in $\xi(\mathbb{T}, T_i, T_j)$ and $e_u^k <_{\mathbb{T}} e_u^m <_{\mathbb{T}} e_v^l$.*

Definition 24 (Chain Consistency). *An isolated trace \mathbb{T} is chain-consistent if given any $\xi(\mathbb{T}, T_i, T_j)$ is chain-isolated and self-contained (for some $T_i, T_j \in \mathbb{T}$).*

Finally, *harmony* is satisfied if the preceding properties are satisfied within the entire trace.

Definition 25 (Harmony). *Trace \mathbb{T} is harmonious iff it satisfies all of the following: a) minimalism, b) consonance, c) obbligato, d) coherence, commit accord, abort accord, and abort coda, e) isolation, f) decisiveness, g) chain consistency, h) unique writes.*

5.3 Last-use Opacity from Harmony

Theorem 1 (Harmonious Trace Last-use Opacity). *Given harmonious H , s.t. $H = \text{hist}(\mathbb{T})$, if \mathbb{T} is harmonious, H is last-use opaque.*

6 OptSVA Harmony

Let $\bar{\mathbb{T}}$ be any trace produced by OptSVA.

Observation 1 (Memory Access Pattern). *OptSVA generates view and update events for variable x precisely as a result of executing the following lines:*

- in procedure **checkpoint** at line 99—view event,
- in procedure **read_buffer** at line 29—view event,
- in procedure **write_buffer** at line 56—routine update event,
- in procedure **commit** at line 113—routine update event,
- in procedure **abort** at line 90—recovery update event.

Observation 2 (Closing Write Identification). *If after executing a write operation on x by T_i it is true that $\text{wub}_i(x) = \text{wc}_i(x)$, then that is the closing write operation execution on x in T_i .*

Lemma 4 (Version Order). *Any two transactions $T_i, T_j \in \bar{\mathbb{T}}$ s.t. $\text{ASet}_i \cap \text{ASet}_j \neq \emptyset$ are isolation ordered: if $\exists x \in \text{ASet}_i \cap \text{ASet}_j$ $\text{pv}_i(x) < \text{pv}_j(x)$, then $\forall x, y \in \text{ASet}_i \cap \text{ASet}_j$, $\text{pv}_i(x) < \text{pv}_j(x)$.*

Proof. During **start** every transaction acquires a value of $\text{pv}_i(x)$. Since the acquisition is guarded by locks, it is performed atomically, so that if transaction T_i , starts acquiring $\forall x \in \text{ASet}_i$, $\text{pv}_i(x)$, then no other T_j acquires $\forall x \in \text{ASet}_j \cap \text{ASet}_i$, $\text{pv}_i(x)$ until transaction T_i completes acquiring and releases the locks. Hence, if for any two T_i, T_j , if $\exists x \in \text{ASet}_i \cap \text{ASet}_j$ $\text{pv}_i(x) < \text{pv}_j(x)$, then $\forall x, y \in \text{ASet}_i \cap \text{ASet}_j$, $\text{pv}_i(x) < \text{pv}_j(x)$. \square

Corollary 2 (Version Order from Isolation Order). *Given transactions T_i, T_j s.t. $T_i \prec_{\bar{\mathbb{T}}} T_j$, then $\forall x \in \text{ASet}_i \cap \text{ASet}_j$, $\text{pv}_j(x) < \text{pv}_i(x)$.*

Lemma 5 (Minimalism). *$\bar{\mathbb{T}}$ is minimalistic.*

Proof. If x is read-only in T_i , then there is exactly one view event on x in T_i (line 29). If x is not read-only, then there is exactly one view event on x in T_i executed as part of procedure **checkpoint** (line 99), either during the first read, the closing write (in **write_buffer**), or, if not previously invoked, during **commit**.

Routine update events are executed only after the closing write (in **write_buffer**—line 56), so at most once, or during **commit** (line 113), if there were writes, but the upper bound on writes was not reached. Hence, routine update events occur at most once per variable.

A recovery update event can occur only during **abort** line 90, at most once per variable. \square

Lemma 6 (Obligatory Checkpoints). *If T_i issues an update event or a view update event, T_i invoked `checkpoint`.*

Proof. View events are only executed as part of `checkpoint`.

A routine update event is only executed as part of `write_buffer` at line 56, which is dominated by lines 52–53, which executes `checkpoint` if it was not previously executed.

A recovery update event occurs as a result of executing line 90, which is guarded by a condition that $\text{wc}_i(x) > 0$, so a write must have been executed. Furthermore, $\text{pv}_i(x) - 1 > \text{lv}(x)$ must be true, which implies that T_i released x , which means the closing write executed, so `write_buffer` was started asynchronously. That procedure executes a `checkpoint` if it was not executed beforehand at lines 52–53. \square

Lemma 7 (Always View Before Update). *If transaction T_i issues an update event $e_u = s_i(x) \square$ in trace \bar{T} , then there is $e_v = g_i(x) \square \in \bar{T} | T_i$ s.t. $e_v <_{\bar{T}} e_u$.*

Proof. From Lemma 6, if T_i executes an update event, then it executes `checkpoint` before the event is issued. Since `checkpoint` issues a view event, then a view event is issued before an update event. \square

Lemma 8 (Wait at Access). *Given transactions T_i, T_j s.t. $\text{pv}_j(x) < \text{pv}_i(x)$, T_i does not issue a view or update event on x until T_j executes `release` on x , `abort`, or `commit`.*

Proof. Let T_k be such that $\text{pv}_k(x) = \text{pv}_i(x) - 1$. Every invocation of `checkpoint` is dominated by an instruction that waits until the condition $\text{pv}_i(x) - 1 = \text{lv}(x)$: line 19 by line 18, line 109 by line 107, and line 53 by line 49. Since, from Lemma 6, every view or update event is preceded by the invocation of `checkpoint`, then each view or update event is dominated by an instruction that waits until $\text{pv}_i(x) - 1 = \text{lv}(x)$. Hence in order for T_i to issue a event it must be true that $\text{pv}_i(x) - 1 = \text{lv}(x)$.

In order for that condition to be met, some transaction must set $\text{lv}(x)$ to $\text{pv}_i(x) - 1$ (or $\text{pv}_i(x) = 1$, but then there could not be such T_j as assumed). Some transaction T_k modifies a new value of $\text{lv}(x)$ during `release`, `abort`, or `commit` and the value is there set to $\text{pv}_k(x)$. Hence T_i cannot issue any view or update event until some T_k such that $\text{pv}_k(x) = \text{pv}_i(x) - 1$ executes `release`, `abort`, or `commit`.

Every invocation of `release` (by T_k) is dominated by an instruction that waits until the condition $\text{pv}_k(x) - 1 = \text{lv}(x)$ is met: the invocation at line 30 by line 12, and the one at line 57 by line 49. Furthermore, modifying $\text{lv}(x)$ within `commit` (line 70) or `abort` (line 94) also requires that $\text{pv}_k(x) - 1 = \text{lv}(x)$ be first satisfied (at line 93 and line 93, respectively). Hence T_k cannot set $\text{lv}(x)$ to $\text{pv}_k(x)$ view or update event unless $\text{pv}_k(x) = 1$ or until some T_l such that $\text{pv}_l(x) = \text{pv}_k(x) - 1$ executes `release`, `abort`, or `commit`.

Assuming that $\text{pv}_k(x) > 1$, and that some T_l s.t. $\text{pv}_l(x) = \text{pv}_k(x) - 1$ exists, then, since T_i cannot issue any view or update event until T_k sets $\text{lv}(x)$ in `release`, `abort`, or `commit` and since T_k cannot set $\text{lv}(x)$ until T_l executes `release`, `abort`, or `commit`, then T_i cannot issue any view or update events until T_l executes `release`, `abort`, or `commit`. Since $\text{pv}_i(x) - 1 = \text{pv}_k(x)$ and $\text{pv}_k(x) - 1 = \text{pv}_l(x)$ then $\text{pv}_l(x) < \text{pv}_i(x)$.

It follows by induction then that given any T_j s.t. $\text{pv}_j(x) < \text{pv}_i(x)$, T_i does not issue a view or update event on x until T_j executes **release**, **abort**, or **commit**. \square

Lemma 9 (Recovery Versions from Version Order). *Given transactions T_i, T_j s.t. $\text{pv}_j(x) < \text{pv}_i(x)$, if T_j executes **abort** before T_i executes **checkpoint**, $\text{rv}_j(x) \leq \text{rv}_i(x)$. otherwise $\text{rv}_j(x) < \text{rv}_i(x)$.*

Proof. Transaction T_i sets $\text{rv}_i(x)$ to $\text{cv}(x)$ only during **checkpoint** (line 100). Every invocation of **checkpoint** is dominated by an instruction that waits until the condition $\text{pv}_i(x) - 1 = \text{lv}(x)$: line 19 by line 18, line 109 by line 107, and line 53 by line 49.

In order for that condition to be met, some transaction must set $\text{lv}(x)$ to $\text{pv}_i(x) - 1$ (or $\text{pv}_i(x) = 1$, but then there could not be such T_j as assumed, so necessarily $\text{pv}_i(x) > 1$). Some transaction T_k can set a new value of $\text{lv}(x)$ during **release**, **abort**, or **commit**. Hence T_i sets the value of $\text{rv}_i(x)$ only after T_k such that $\text{pv}_k(x) = \text{pv}_i(x) - 1$ executes **release**, **abort**, or **commit**. Thus, since value of $\text{cv}(x)$ is there set by T_k to $\text{pv}_k(x)$ in case of **release** (line 103) and **commit** (line 74), or $\text{rv}_k(x)$ in case of **abort** (line 91), $\text{rv}_i(x) = \text{rv}_k(x)$ if T_k aborts before T_i executes **checkpoint** and $\text{rv}_i(x) = \text{pv}_k(x)$ otherwise.

Since $\text{rv}_k(x)$ either trivially equals 0 if $\text{pv}_k(x) = 1$, or is acquired by analogy from some T_l s.t. $\text{pv}_l(x) = \text{pv}_k(x) - 1$, then $\text{rv}_k(x) \leq \text{rv}_i(x)$.

Furthermore, under the assumption that T_k does not execute **abort** prior to T_i executing **checkpoint**, then value of $\text{cv}(x)$ is there set by T_k only either within **release** or **commit**, and thus $\text{cv}(x) = \text{pv}_k(x)$ during T_i 's **checkpoint**, so $\text{rv}_i(x) = \text{pv}_k(x)$. Since $\text{pv}_k(x) < \text{pv}_i(x)$, then $\text{rv}_k(x) < \text{rv}_i(x)$.

By extension, given T_j s.t. $\text{pv}_j(x) < \text{pv}_i(x)$, either $k = j$, or $\text{pv}_j(x) < \text{pv}_k(x)$.

In the former case, necessarily $\text{rv}_k(x) \leq \text{rv}_i(x)$ if T_j executes **abort** before T_i executes **checkpoint**, or $\text{rv}_k(x) < \text{rv}_i(x)$.

In the latter case, there must be some T_l s.t. $\text{pv}_l(x) = \text{pv}_k(x)$. Then, if T_l executes **abort** before T_k executes **checkpoint**, $\text{rv}_l(x) \leq \text{rv}_k(x)$, otherwise $\text{rv}_l(x) < \text{rv}_k(x)$. Furthermore, either $l = j$, or $\text{pv}_j(x) < \text{pv}_l(x)$.

It then follows by induction that given any T_j s.t. $\text{pv}_j(x) < \text{pv}_i(x)$, if T_j executes **abort** before T_i executes **checkpoint**, $\text{rv}_l(x) \leq \text{rv}_k(x)$, otherwise $\text{rv}_l(x) < \text{rv}_k(x)$. \square

Lemma 10 (Isolation). *Trace $\bar{\mathbb{T}}$ is isolated.*

Proof. Every routine update event, view event, and recovery event is dominated by an access conditions ($\text{pv}_i(x) - 1 = \text{lv}(x)$). This condition is satisfied for T_i if $\text{lv}(x) = 0$ and $\text{pv}_i(x) = 0$, or if some transaction T_j s.t. $\text{pv}_j(x) = \text{pv}_i(x) - 1$ releases x by setting $\text{lv}(x)$ to $\text{pv}_j(x)$ during **commit** or after closing write or after the first non-local read (and thus after any $\text{os}_j(x)$ or $g_j(x)$). \square

Since events are guarded by access conditions, since variables are released after all view or routine update events are issued by a transaction, and since each transactions are version-ordered, then for any T_i, T_j , $\exists x \in \text{ASet}_i \cap \text{ASet}_j$ if $\exists e_i = \text{os}_i(x) \in \bar{\mathbb{T}}|T_i$ or $e_i = g_i(x) \in \bar{\mathbb{T}}|T_i$ and $\exists e_j = \text{os}_j(x) \in \bar{\mathbb{T}}|T_j$ or $e_j = g_j(x) \in \bar{\mathbb{T}}|T_j$, and $e_i <_{\mathbb{T}} e_j$ then $\forall y \in \text{ASet}_i \cap \text{ASet}_j$, if $\exists e'_i = \text{os}_i(y) \in \bar{\mathbb{T}}|T_i$ or $e'_i = g_i(y) \in \bar{\mathbb{T}}|T_i$ and $\exists e'_j = \text{os}_j(y) \in \bar{\mathbb{T}}|T_j$ or $e'_j = g_j(y) \in \bar{\mathbb{T}}|T_j$, and $e'_i <_{\mathbb{T}} e'_j$. \square

Corollary 3 (Isolation Order). *Trace $\bar{\mathbb{T}}$ is isolation-ordered.*

Lemma 11 (Write Consonance). *Any (complete) write operation in $\bar{\mathbb{T}}$ is consonant.*

Proof. Each write is guarded by the condition at line 41, which aborts the transaction if the value that is supposed to be written is not within the domain of the variable. Thus, each write is consonant. \square

Lemma 12 (Routine Update Consonance). *Any routine update event in $\bar{\mathbb{T}}$ is consonant.*

Proof. A routine update event $\circ s_i(x)v$ occurs either as a result of executing a closing write operation on x (line 56) or T_i committing (line 113), if the transaction executed writes, but the upper bound for writes was not reached for x . Clearly, then, if there was a routine update event, then T_i executed a write operation on x . In both cases above $v = \text{buf}_i(x)$ and $\text{buf}_i(x)$ can be set by any write operation, the first non-local read operation, or during start for read-only variables. If there was a write, then x is not read-only, and the first non-local read cannot follow a write, so $\text{buf}_i(x)$ is set within (the most recent) write operation executed by T_i and corresponds to the value written by that operation. Thus, $\forall T_i, \forall e_u^i = \circ s_i(x)v \in \bar{\mathbb{T}}|T_i, \exists op_i = w_i(x)v \rightarrow ok_i \in \bar{\mathbb{T}}|T_i$ s.t. $op_i \rightsquigarrow e_u^i$. Therefore, e_u^i is consonant. \square

Lemma 13 (View Consonance). *Any view event in $\bar{\mathbb{T}}$ is consonant.*

Proof. If a view event occurs, it views the current state of a variable. So given transaction T_i , if there is a view event $e_v = g_i(x)v \in \bar{\mathbb{T}}|T_i$, v corresponds to the current state of x . The only way to change the state of x is via an update event on x . Thus, trivially, for some T_j , if there is $e_u = \circ s_j(x)v' \in \bar{\mathbb{T}}|T_j$ or $e_a = \diamond s_j(x)v' \in \bar{\mathbb{T}}|T_j$, if either e_u or e_a precede e_v so that no other update event on x occurs between either e_u or e_a and e_v , then $v = v'$. Furthermore, from unique routine updates, there cannot be any e_u s.t. $v' = 0$, and since x is initially 0, then $\nexists e_u$ s.t. $v' = 0$ and $e_u < e_v$. \square

Lemma 14 (Local Read Consonance). *Any local read operation execution in $\bar{\mathbb{T}}$ is consonant.*

Proof. If transaction executes a local read on x , then it previously executed a write operation on x , so $\text{wc}_i(x) > 0$. Thus, the read procedure returns at line 25, returning $\text{buf}_i(x)$. The value of $\text{buf}_i(x)$ can be set by any write operation, the first non-local read operation, or during start for read-only variables. If there was a write, then x is not read-only, and the first non-local read cannot follow a write, so $\text{buf}_i(x)$ is set within (the most recent) write operation executed by T_i and corresponds to the value written by that operation. Thus, $\forall T_i, \forall op^i = r_i(x) \rightarrow v \in \bar{\mathbb{T}}|T_i$ if op_i is local, $\exists op'_i = w_i(x)v' \rightarrow ok_i \in \bar{\mathbb{T}}|T_i$ s.t. $v' = v$. Therefore, op_i is consonant. \square

Lemma 15 (Non-local Read Consonance). *Any non-local read operation execution in $\bar{\mathbb{T}}$ is consonant.*

Proof. If x is read-only in T_i , then during **start**, a view event occurs within **read_buffer** (line 29), and the state of x is saved in $\text{buf}_i(x)$. Then, subsequent writes return the value of $\text{buf}_i(x)$ (line 25) (waiting if necessary). Thus, they depend on that view event.

Otherwise, a non-local read operation on x is one that is not preceded by a write on x , so $\text{wc}_i(x) = 0$. The first such read executes **checkpoint** which initiates a view event (line 99). The value obtained by that event is saved in $\text{st}_i(x)$ and later $\text{buf}_i(x)$ is set to the same value. Finally, that value is returned at line 25. Subsequent non-local reads use the same value stored in $\text{buf}_i(x)$. The value remains unchanged, since it only be overwritten by a write, but the occurrence of a preceding write would mean the read is local (and since x is not read-only, and there was a preceding non-local read). Thus, all non-local reads depend on the view event issued during **checkpoint**.

Thus, $\forall T_i, \forall \text{op}^i = r_i(x) \rightarrow v \in \bar{\mathbb{T}}|T_i$ if op_i is non-local, $\exists e_v^i = g_i(x)v' \in \bar{\mathbb{T}}|T_i$ s.t. $v' = v$. \square

Lemma 16 (Conservative Recovery Update Events). *Any recovery update event in $\bar{\mathbb{T}}$ is conservative.*

Proof. The recovery update event in T_i occurs as a result of executing line 90, which updates the state of x to $\text{st}_i(x)$. This is done only if $\text{rv}_i(x) \neq \text{cv}(x)$ and $\text{wc}_i(x) > 0$. If is to be true, Since $\text{rv}_i(x)$ is set to the value of $\text{cv}(x)$ only during **checkpoint** and **read_buffer**, and since the requirement that $\text{wc}_i(x) > 0$ excludes the latter, this condition checks whether the current transaction previously made a checkpoint. Executing **checkpoint** entails a view event that sets $\text{st}_i(x)$ to the current value of x . Hence, if $e_a = \blacklozenge s_i(x)v \in \bar{\mathbb{T}}|T_i$ then there exists $e_v = g_i(x)v \in \bar{\mathbb{T}}|T_i$ s.t. $e_a < e_v$. \square

Lemma 17 (Clean Recovery Update Events). *Any recovery update event in $\bar{\mathbb{T}}$ is clean.*

Proof. Assume by contradiction that there exists $e_a = \blacklozenge s_i(x)v$ in $\bar{\mathbb{T}}|T_i$ and $e_v = \text{getixv}$ that justifies that e_s is conservative, and $e'_a = \blacklozenge s_j(x)v'$ in $\bar{\mathbb{T}}|T_j$ s.t. $T_j \prec_{\bar{\mathbb{T}}}^x T_i$ and $e_v <_{\bar{\mathbb{T}}} e'_a <_{\bar{\mathbb{T}}} e_a$. This implies that T_j executes **abort** (and satisfies the condition $\text{rv}_j(x) = \text{cv}(x)$) between the point at which T_i executes **checkpoint** and **abort**. If that is the case, as a result of executing **abort**, T_j sets $\text{cv}(x)$ to $\text{rv}_j(x)$, and the.

Given that $T_j \prec_{\bar{\mathbb{T}}}^x T_i$, then $\text{pv}_j(x) < \text{pv}_i(x)$. Since any execution of **checkpoint** for some T_k is guarded by the condition $\text{pv}_k(x) - 1 = \text{lv}(x)$, then T_j executes **checkpoint** before T_i . Hence, T_j acquires $\text{rv}_j(x)$ from $\text{cv}(x)$ before T_i acquires $\text{rv}_i(x)$ from $\text{cv}(x)$.

The value of $\text{rv}_j(x)$ is equal to the value of $\text{cv}(x)$ at the point when T_j executed **checkpoint** (i.e. when $\text{pv}_i(x) - 1 = \text{lv}(x)$). The value of $\text{cv}(x)$ is set to $\text{pv}_k(x)$ when T_k executes **release** or **commit** or to $\text{rv}_k(x)$ when T_k aborts. Thus, when T_j executes **checkpoint**, since $\text{pv}_i(x) - 1 = \text{lv}(x)$, then either:

- a) $\text{cv}(x) = \text{pv}_k(x) = \text{pv}_j(x) - 1$ (if T_k released x or committed),
- b) $\text{cv}(x) = \text{rv}_k(x)$ and $\text{rv}_k(x) < \text{pv}_j(x)$ (if T_k aborted), or
- c) $\text{cv}(x) = 0$ (if there is no such T_k).

In any case, $\mathbf{rv}_j(x) < \mathbf{pv}_i(x)$.

T_i is capable of executing **checkpoint** after T_j commits, aborts, or releases x . Since T_j executes **abort** between T_i 's **checkpoint** and **abort**, then only the third option remains. If T_j executes **release** for x , then it sets $\mathbf{cv}(x)$ to $\mathbf{pv}_j(x)$. Following the logic from the previous paragraph, this means that when T_i assigns $\mathbf{cv}(x)$ to $\mathbf{rc}_i(x)$, $\mathbf{pv}_j(x) \leq \mathbf{cv}(x)$, so $\mathbf{pv}_j(x) \leq \mathbf{rv}_i(x)$, so $\mathbf{rv}_j(x) < \mathbf{rv}_i(x)$.

Hence, after $\mathbf{cv}(x)$ to $\mathbf{rv}_j(x)$ during abort, it is not true that $\mathbf{cv}(x) = \mathbf{rv}_j(x)$. Thus, e_a cannot occur once e'_a occurs, which is a contradiction. \square

Lemma 18 (Needed Recovery Update Events). *Any recovery update event in \mathbb{T} is needed.*

Proof. The recovery update event occurs as a result of executing line 90, which is guarded by a condition that $\mathbf{wc}_i(x) > 0$, so a write must have been executed. Furthermore, $\mathbf{pv}_i(x) - 1 > \mathbf{lv}(x)$ must be true, which implies that T_i released x , which means the closing write executed, so **write_buffer** was started asynchronously. If that is the case, the recovery update event cannot execute until **write_buffer**, which means a routine update event on x will have executed before the recovery update event on x . \square

Lemma 19 (Dooming Recovery Update Events). *Any recovery update event in \mathbb{T} is dooming.*

Proof. Trivially, since any recovery update event occurs only within **abort**. \square

Lemma 20 (Ending Recovery Update Events). *Any recovery update event in \mathbb{T} is ending.*

Proof. Trivially, since any recovery update event occurs only within **abort**, and there are no other update or view events on the same variable in **abort**. \square

Lemma 21 (Recovery Update Consonance). *Any recovery update event in \mathbb{T} is consonant.*

Proof. Since each recovery update is conservative (from Lemma 16), needed (from Lemma 18), dooming (from Lemma 19), ending (from Lemma 20), and clean (from Lemma 17), then each recovery update is consonant. \square

Lemma 22 (Trace Consonance). *Trace $\bar{\mathbb{T}}$ is consonant.*

Proof. From Lemmata 11–15 and 21. \square

Lemma 23 (Comitted Write Obbligato). *Given T_i that is committed in $\bar{\mathbb{T}}$, every non-local write operation execution $op_i = w_i(x)v \rightarrow ok_i \in \bar{\mathbb{T}}|T_i$ is in committed obbligato.*

Proof. If T_i executes a write corresponding to op_i , then, if at the end of the execution it is true that $\mathbf{wc}_i(x) = \mathbf{wub}_i(x)$, **write_buffer** is executed, which causes a routine update event to execute, writing the value of $\mathbf{buf}_i(x)$ to x .

Since $\mathbf{wc}_i(x) = \mathbf{wub}_i(x)$ no other writes follow, and since x is not read-only in T_i , then the value written to x in **write_buffer** is the value passed to the write operation. In that case there is $e_u = \circ s_i(x)v \in \bar{\mathbb{T}}|T_i$. Since commit will not return until **write_buffer** finishes executing, then trivially $inv_i(w_i(x)v) <_{\mathbb{T}|T_i} e_u <_{\mathbb{T}|T_i} res_i(C_i)$.

If it is true that $\mathbf{wc}_i(x) = \mathbf{wub}_i(x)$, then **write_buffer** is not executed, but during **commit**, the same condition is checked again, and if it is not satisfied, T_i writes the value from $\mathbf{buf}_i(x)$ to x . Thus, by analogy to the paragraph above, there is $e_u = \circ s_i(x)v \in \bar{\mathbb{T}}|T_i$. Since this is executed within **commit**, then $\text{inv}_i(w_i(x)v) <_{\bar{\mathbb{T}}|T_i} e_u <_{\bar{\mathbb{T}}|T_i} \text{res}_i(C_i)$. \square

Lemma 24 (Closing Write Obligato). *Given T_i that is decided on x in $\bar{\mathbb{T}}$, every non-local write operation execution $op_i = w_i(x)v \rightarrow ok_i \in \bar{\mathbb{T}}|T_i$ is in closing write obligato.*

Proof. If T_i executes a write corresponding to op_i , then, at the end of the execution, if op_i is a closing write it is necessarily true that $\mathbf{wc}_i(x) = \mathbf{wub}_i(x)$. This causes **write_buffer** to be executed, line 49 which causes a routine update event to execute, writing the value of $\mathbf{buf}_i(x)$ to x .

Since $\mathbf{wc}_i(x) = \mathbf{wub}_i(x)$ no other writes follow, and since x is not read-only in T_i , then the value written to x in **write_buffer** is the value passed to the write operation. In that case there is $e_u = \circ s_i(x)v \in \bar{\mathbb{T}}|T_i$. Since **commit** will not return until **write_buffer** finishes executing, then trivially $\text{inv}_i(w_i(x)v) <_{\bar{\mathbb{T}}|T_i} e_u <_{\bar{\mathbb{T}}|T_i} \text{res}_i(C_i)$. Hence $op_i \rightsquigarrow e_u$. T_i executes **release** only following issuing e_u **wb:release**.

If $T_i <_{\bar{\mathbb{T}}} T_j$, then $\mathbf{pv}_i(x) < \mathbf{pv}_j(x)$ (Corollary 2). From Lemma 8, to issue any $e_v = g_j(x)\square$, since $\mathbf{pv}_i(x) < \mathbf{pv}_j(x)$, T_i must execute **abort**, **commit**, or **release**. Hence T_j does not issue e_v before T_i executes **release**, which requires that T_i issues e_u so that $e_u <_{\bar{\mathbb{T}}} e_v$. \square

Lemma 25 (View Write Obligato). *Given $T_i \in \bar{\mathbb{T}}$, if $\exists T_j \in \mathbb{T}$, s.t. $T_i <_{\bar{\mathbb{T}}} T_j$, if there is $op_i = w(i)x\square ok_i \in \bar{\mathbb{T}}|T_i$, and $e_v = g_j(x)\square \in \bar{\mathbb{T}}|T_j$, then op_i is in view write obligato.*

Proof. If $T_i <_{\bar{\mathbb{T}}} T_j$, then $\mathbf{pv}_i(x) < \mathbf{pv}_j(x)$ Corollary 2. From Lemma 8 e_v occurs only after T_i releases x , commits, or aborts. Since according to the assumption, T_i cannot abort prior to T_j issuing e_v , T_i either releases x or commits prior to T_j issuing e_v .

If T_i releases x it executes **release**. This can occur as a result of executing line 30 or line 57. Since T_i executes op_i , then line 30 cannot be executed, since it can only be reached if T_i only ever reads x (condition at line 10). Hence T_i must execute line 57, which is dominated by line 56, which issues a write event $e_u = s_i(x)v$, where v is the value of $\mathbf{buf}_i(x)$.

Since **release** was executed at line 57, **write_buffer** must have been executed at line 49. Then the value written to x in **write_buffer** is the value passed to the write operation. In that case there is $e_u = \circ s_i(x)v \in \bar{\mathbb{T}}|T_i$. Since **commit** will not return until **write_buffer** finishes executing, then trivially $\text{inv}_i(w_i(x)v) <_{\bar{\mathbb{T}}|T_i} e_u <_{\bar{\mathbb{T}}|T_i} \text{res}_i(C_i)$. Hence $op_i \rightsquigarrow e_u$. T_i executes **release** only following issuing e_u **wb:release**. \square

Lemma 26 (Obligato). *Trace $\bar{\mathbb{T}}$ is obligato.*

Proof. From Lemmas 23, 24, and 25. \square

Lemma 27 (Decisiveness). *Trace $\bar{\mathbb{T}}$ is decisive.*

Proof. If $T_j \dot{\sim} T_i$, then for any $x \in \text{ASet}_i \cap \text{ASet}_j$, $\text{pv}_i(x) < \text{pv}_j(x)$ (Lemma 10). Before any view event occurs, T_j must pass the condition $\text{pv}_k(x) - 1 = \text{lv}(x)$ in `read_buffer` or `checkpoint`. Hence, before any $e_v = g_j(x)v$ can occur, some T_k s.t. $\text{pv}_i(x) - 1 = \text{pv}_k(x)$ must set $\text{pv}_k(x)$ to $\text{lv}(x)$. Transaction T_j issues a routine update event $e_u = \circ s_j(x)v'$, whenever it commits or releases x . If it releases, it means that $\text{wc}_j(x) = \text{wub}_j(x)$, which implies that op_i is closing. Otherwise, T_j will update on commit, meaning that it will issue $\text{res}_j(C_j)$ once it returns from the `commit` procedure. Before returning from the closing write or commit, T_j sets $\text{lv}(x)$ to $\text{pv}_j(x)$. In either case, this happens only afterward e_u is issued. Since there is no waiting between e_u and either a commit or a last write returning, no other transaction may execute anything on x in the meantime. Thus, any transaction T_k s.t. $\text{pv}_k(x)$ that waits until $\text{pv}_k(x) - 1 = \text{lv}(x)$ and $\text{pv}_k(x) - 1 = \text{pv}_j(x)$ will wait until T_j returns from the closing write or commit procedure, and so will any subsequent transactions according to version order. Thus, if $\text{pv}_i(x) < \text{pv}_j(x)$ and T_j commits, then either op_j is closing or $e_u <_{\bar{\mathbb{T}}} \text{res}_j(C_j) < e_v$. \square

Lemma 28 (Abort Accord). *Trace $\bar{\mathbb{T}}$ is in abort accord.*

Proof. Let T_i, T_j be two transactions in $\bar{\mathbb{T}}$ s.t.

a) $T_j \dot{\sim} T_i$, T_i is aborted in $\bar{\mathbb{T}}$.

Assume by contradiction that T_j commits in $\bar{\mathbb{T}}$, meaning it executes `commit` successfully. Thus it passes $\forall x, \text{cv}(x) \geq \text{rv}_j(x)$.

Since $T_j \dot{\sim} T_i$, $\exists e_v = g_j(x)v \in \bar{\mathbb{T}}|T_j$ and $\exists e_u = \circ s_i(x)v \in \bar{\mathbb{T}}|T_i$.

If T_i aborted before e_v was issued, then from abort coda, $\exists e_a = \blacklozenge s_k(x) \square$ in some T_k that precedes the abort, which contradicts that $T_j \dot{\sim} T_i$. Hence T_i aborts only after e_v is issued.

Since $T_j \dot{\sim} T_i$, then $T_i <_{\bar{\mathbb{T}}}^x T_j$, so from Corollary 2, $\text{pv}_i(x) < \text{pv}_j(x)$. From Lemma 8, e_v cannot occur until T_i aborts, commits, or releases x . Since T_i aborts after e_v , then it must therefore release x prior to e_v .

Since $\text{pv}_i(x) < \text{pv}_j(x)$, then from Lemma 9, $\text{rv}_i(x) < \text{rv}_j(x)$. When T_i aborts, it sets $\text{cv}(x)$ to $\text{rv}_i(x)$. From coherence, T_j commits after T_i aborts. Thus, when T_j commits, $\text{cv}(x) = \text{rv}_i(x)$, so since $\text{rv}_i(x) < \text{rv}_j(x)$, then $\text{cv}(x) < \text{rv}_j(x)$, which contradicts the condition that $\text{cv}(x) \geq \text{rv}_j(x)$.

Thus T_j cannot commit.

b) $\exists e_u = \circ s_i(x) \square \in \bar{\mathbb{T}}|T_i$ and $e = \circ s_j(x) \square \in \bar{\mathbb{T}}|T_j$ or $e = g_j(x) \square \in \bar{\mathbb{T}}|T_j$ and $e_a = \blacklozenge s_i(x) \square \in \bar{\mathbb{T}}|T_i$, and $e_u <_{\bar{\mathbb{T}}} e <_{\bar{\mathbb{T}}} e_a$.

Assume by contradiction that T_j commits in $\bar{\mathbb{T}}$, meaning it executes `commit` successfully. Thus it passes $\forall x, \text{cv}(x) \geq \text{rv}_j(x)$.

Since $e_u <_{\bar{\mathbb{T}}} e$, then from isolation it follows that $T_i <_{\bar{\mathbb{T}}}^x T_j$. Hence, from Corollary 2, $\text{pv}_i(x) < \text{pv}_j(x)$. Since e_a must be issued during `abort`, then from coherence, T_j cannot commit prior to e_a occurring. Furthermore, T_j cannot commit until T_i returns from `abort`.

If T_i returned from `abort`, then it executed line 91, so $\text{cv}(x) = \text{rv}_i(x)$ prior to T_j committing.

From Lemma 9, since $\text{pv}_i(x) < \text{pv}_j(x)$, then $\text{rv}_i(x) < \text{rv}_j(x)$. When T_i aborts, it sets $\text{cv}(x)$ to $\text{rv}_i(x)$. From coherence, T_j commits after T_i aborts. Thus, when T_j commits, $\text{cv}(x) = \text{rv}_i(x)$, so since $\text{rv}_i(x) < \text{rv}_j(x)$, then $\text{cv}(x) < \text{rv}_j(x)$, which contradicts the condition that $\text{cv}(x) > \text{rv}_j(x)$.

Thus T_j cannot commit.

□

Lemma 29 (Commit Accord). *Trace $\bar{\mathbb{T}}$ is in commit accord.*

Proof. Let T_i, T_j be transaction in \mathbb{T} s.t. $T_j \rightsquigarrow T_i$ and T_j is committed in $\bar{\mathbb{T}}$.

Let us assume by contradiction that T_i is not committed in $\bar{\mathbb{T}}$. So T_i is either aborted or live in $\bar{\mathbb{T}}$. From coherence, if T_j cannot commit until T_i commits or aborts. Thus T_i is not live in $\bar{\mathbb{T}}$, so it is aborted in $\bar{\mathbb{T}}$.

Since $T_j \rightsquigarrow T_i$, $\exists e_v = g_j(x)v \in \bar{\mathbb{T}}|T_j$ and $\exists e_u = \circ s_i(x)v \in \bar{\mathbb{T}}|T_i$.

If T_i aborted before e_v was issued, then from abort coda, $\exists e_a = \blacklozenge s_k(x) \square$ in some T_k that precedes the abort, which contradicts that $T_j \rightsquigarrow T_i$. Hence T_i aborts only after e_v is issued.

Since $T_j \rightsquigarrow T_i$, then $T_i \prec_{\mathbb{T}}^x T_j$, so from Corollary 2, $\text{pv}_i(x) < \text{pv}_j(x)$. From Lemma 8, e_v cannot occur until T_i aborts, commits, or releases x . Since T_i aborts after e_v , then it must therefore release x prior to e_v .

Since $\text{pv}_i(x) < \text{pv}_j(x)$, then from Lemma 9, $\text{rv}_i(x) < \text{rv}_j(x)$. When T_i aborts, it sets $\text{cv}(x)$ to $\text{rv}_i(x)$. From coherence, T_j commits after T_i aborts. Thus, when T_j commits, $\text{cv}(x) = \text{rv}_i(x)$, so since $\text{rv}_i(x) < \text{rv}_j(x)$, then $\text{cv}(x) < \text{rv}_j(x)$, which contradicts the condition that $\text{cv}(x) > \text{rv}_j(x)$.

Thus T_i cannot abort.

□

Lemma 30 (Abort Coda). *Trace $\bar{\mathbb{T}}$ has coda.*

Proof. a) If T_i aborts in $\bar{\mathbb{T}}$ (so $r = \text{res}_i(A_i) \in \bar{\mathbb{T}}|T_i$) then if $\exists e_u = \circ s_x(v) \in \bar{\mathbb{T}}|T_i$ then for some T_j (possibly $i = j$) s.t. $j = i$ or $T_j \prec_{\mathbb{T}}^x T_i$, $\exists e_a = \blacklozenge s_j(x) \square \in \bar{\mathbb{T}}|T_j$ s.t. $e_u <_{\bar{\mathbb{T}}} e_a <_{\bar{\mathbb{T}}} r$.

If there is such e_u , then T_i executes **checkpoint** (Lemma 6). Since there is such e_u there is also a write operation execution on x in $\bar{\mathbb{T}}|T_i$ (Lemma 12), so $\text{wc}_i(x) > 0$ (line 46).

If there is such e_u , then T_i executes **release** for x or **commit**. Since T_i aborts, then **commit** is not possible, so T_i executes **release** for x . Therefore T_i sets $\text{lv}(x)$ to $\text{pv}_x(i) - 1$.

- i) If no other transaction modified $\text{cv}(x)$ between the point at which T_i executed **checkpoint** and **abort**, then $\text{cv}(x) = \text{rv}_i(x)$, thus during abort T_i satisfies the condition on line 87 and executes line 90, issuing the recovery event e_a . Since e_a is issued during **abort**, then $e_u <_{\bar{\mathbb{T}}} e_a <_{\bar{\mathbb{T}}} r$.
- ii) If there is T_j s.t. T_j modifies $\text{cv}(x)$ between the points at which T_i executed **checkpoint** and **abort**, s.t. $T_j \prec_{\mathbb{T}}^x T_i$, then T_j executes **release**, **commit**, or **abort** between the points at which T_i executed **checkpoint** and **abort**. For the sake of simplicity we assume that there is no other T_k that modifies $\text{cv}(x)$ between those two points s.t. $T_k \prec_{\mathbb{T}}^x T_i$.

Since T_i executes **checkpoint**, it issues a view event $e_v = g_i(x) \square$. In addition, since $T_j \prec_{\mathbb{T}}^x T_i$, then from Corollary 2, $\text{pv}_j(x) < \text{pv}_i(x)$. From

Lemma 8, e_v cannot occur until T_j aborts, commits, or releases x . Since T_j is supposed to execute **release**, **abort**, or **commit** after T_i executes **checkpoint**, hence after e_v , then T_j must therefore release x prior to e_v . Hence T_j executes **commit**, or **abort** between the points at which T_i executed **checkpoint** and **abort**.

If T_i executes **commit**, then in order to set $cv(x)$ to $pv_j(x)$, it must be true that $rv_i(x) = cv(x)$. But if T_j executed **release**, then $cv(x)$ is set to $pv_j(x)$. Since $rv_x(j) \neq pv_j(x)$, then $rv_i(x) \neq cv(x)$, so T_j cannot set $cv(x)$ as a result of a commit. Hence, T_j executes **abort** between the points at which T_i executed **checkpoint** and **abort**.

If T_j executes **abort**, then this implies that T_j executes line 91, and therefore also line 90, thus tr_j issues recovery event e_a during **abort**. Thus, $e_u <_{\bar{\mathbb{T}}} e_a <_{\bar{\mathbb{T}}} r$.

- b) If T_i commits in $\bar{\mathbb{T}}$ (so $r = res_i(C_i) \in \bar{\mathbb{T}}|T_i$) then if $\exists e = \circ s_x(v) \in \bar{\mathbb{T}}|T_i$ or $e = g_x(v) \in \bar{\mathbb{T}}|T_i$ then for no T_j s.t. $j = i$ or $T_j \dot{<}_{\bar{\mathbb{T}}} T_i$, $\exists e_a = \blacklozenge s_j(x) \square \in \bar{\mathbb{T}}|T_j$ s.t. $e_u <_{\bar{\mathbb{T}}} e_a <_{\bar{\mathbb{T}}} r$.

If there is such e , then T_i executes **checkpoint** (Lemma 6). If T_i successfully commits, then means that T_i passes the condition for x that $cv(x) \geq rv_i(x)$.

Assume by contradiction that there is such e_a in some T_j . Since $e_a \in \bar{\mathbb{T}}|T_j$, then T_j must execute line 90, which also means that it executes line 91 and therefore sets $cv(x)$ to $rv_j(x)$.

Since $T_j \dot{<}_{\bar{\mathbb{T}}} T_i$, then from Corollary 2 $pv_j(x) < pv_i(x)$ and from Lemma 9, $rv_j(x) < rv_i(x)$. Thus, $cv(x) < rv_i(x)$ which contradicts that $cv(x) \geq rc_i(x)$. Thus there is no such T_j . \square

Lemma 31 (Coherence). *Trace $\bar{\mathbb{T}}$ is coherent.*

Proof. If $T_i \dot{<}_{\bar{\mathbb{T}}} T_j$, then $pv_i(x) < pv_j(x)$. In order to commit or abort, any T_k must pass the condition $pv_k(x) - 1 = ltv(x)$. In addition, each T_k sets $ltv(x)$ to $pv_k(x)$ only at the end of either committing or aborting. Hence, if T_k cannot commit or abort until some T_l s.t. $pv_k(x) - 1 = pv_l(x)$ finishes committing or aborting. Hence if T_j committed, it must have passed the condition $pv_k(x) - 1 = ltv(x)$, and since $pv_i(x) < pv_j(x)$, T_i must have committed or aborted before T_j committed. Thus, given $r_j = res_j(C_j) \in \bar{\mathbb{T}}|T_j$, then there is $r_i = res_i(C_i) \in \bar{\mathbb{T}}|T_i$ or $r_i = res_i(A_i) \in \bar{\mathbb{T}}|T_i$ and $r_i <_{\bar{\mathbb{T}}} r_j$. \square

Lemma 32 (Chain Isolation). *Given trace $\bar{\mathbb{T}}$ and transactions $T_i, T_j \in \bar{\mathbb{T}}$ s.t. there is $\xi(\bar{\mathbb{T}}, T_i, T_j)$, $\forall T_k \in \xi(\bar{\mathbb{T}}, T_i, T_j)$ s.t. $e_u^k = \circ s_k(x)v$, there is no T_l s.t. $\exists e_a^l = \blacklozenge s_l(x)v'$ where $v = v'$ and e^l is between e^k and any other event in any transaction in $\xi(\bar{\mathbb{T}}, T_i, T_j)$.*

Proof. Assume by contradiction that there exists T_l such that $\exists e^l = \blacklozenge s_l(x)v'$ and e_a^l is between e_u^k and any other event $e \in \bar{\mathbb{T}}|\xi(\bar{\mathbb{T}}, T_i, T_j)$. This means that either $e \in \bar{\mathbb{T}}|T_l$ and $e_u^k <_{\bar{\mathbb{T}}} e_a^l <_{\bar{\mathbb{T}}} e$, or $\exists T_n$ s.t. $e \in \bar{\mathbb{T}}|T_n$.

- a) Assume $e \in \bar{\mathbb{T}}|T_l$ and $e_u^k <_{\bar{\mathbb{T}}} e_a^l <_{\bar{\mathbb{T}}} e$.

From Lemma 7 there is a view event $e_v^k = g_k(x) \square \in \bar{\mathbb{T}}|T_k$, and from minimalism there is only one such event in $\bar{\mathbb{T}}|T_k$, so e must be a recovery event

$e = \diamond s_k(x)v$. If T_l executes e_a^l , then from Lemma 18, $\exists e_u^l = \circ s_l(x) \square$ in $\bar{\mathbb{T}}|T_l$ s.t. $e_u^k <_{\bar{\mathbb{T}}} e_a^l$. Hence either $e_u^l <_{\bar{\mathbb{T}}} e_u^k$ or $e_u^k <_{\bar{\mathbb{T}}} e_u^l$. So either $T_l \dot{<}_{\bar{\mathbb{T}}}^x T_k$ or $T_k \dot{<}_{\bar{\mathbb{T}}}^x T_l$.

If $T_l \dot{<}_{\bar{\mathbb{T}}}^x T_k$, then $e_u^l <_{\bar{\mathbb{T}}} e_u^k$, so from Lemma 8, e_u^k cannot occur until e_u^l executes **release**, **commit**, or **abort**, and since $e_u^k <_{\bar{\mathbb{T}}} e_a^l$, then only **release** is viable.

If $T_k \dot{<}_{\bar{\mathbb{T}}}^x T_l$, then from version order $\text{pv}_l(x) < \text{pv}_k(x)$, so from Lemma 9, $\text{rv}_l(x) < \text{rv}_k(x)$. In order for T_l to issue e_a , it must execute **abort** and satisfy the condition line 87. This means that line 91 is executed, so $\text{cv}(x) = \text{rv}_l(x)$.

If subsequently T_k issues e_a , then it must also satisfy the condition at line 87, so $\text{rv}_k(x) = \text{cv}(x)$. But since $\text{rv}_l(x) < \text{rv}_k(x)$, then $\text{cv}(x) < \text{rv}_k(x)$, which contradicts that $\text{cv}(x) = \text{rv}_l(x)$.

The execution of a recovery event on x by T_l is dominated by line 84, which cannot be passed until $\text{pv}_l(x) - 1 = \text{ltv}(x)$. Any transaction T_n sets $\text{ltv}(x)$ to $\text{pv}_n(x)$ as a last action during **commit** (line 80) or **abort** (line 95). Hence T_l cannot proceed to abort until T_n finishes committing or aborting. Since T_n cannot execute line 80 or line 95 if line 68 or line 84 was passed, then T_n cannot proceed to commit or abort until some other T_m s.t. $\text{pv}_n(x) - 1 = \text{pv}_m(x)$ committed or aborted. Hence T_l cannot execute a recovery event until any T_m s.t. $\text{pv}_m(x) < \text{pv}_l(x)$ committed or aborted.

If $T_k \dot{<}_{\bar{\mathbb{T}}}^x T_l$, then from version order $\text{pv}_k(x) < \text{pv}_l(x)$. Hence, T_l executes any events **abort** only after T_k returns from **abort** or **commit**. Hence if T_k executes e_a^k , then $e_a^k <_{\bar{\mathbb{T}}} e_a^l$, which contradicts that $e_a^l <_{\bar{\mathbb{T}}} e_a^k$.

Thus, regardless of whether $T_l \dot{<}_{\bar{\mathbb{T}}}^x T_k$ or $T_k \dot{<}_{\bar{\mathbb{T}}}^x T_l$ there is a contradiction. Therefore, T_l cannot issue such e_a^l between e_u^k and another event in $\bar{\mathbb{T}}|T_l$.

b) Assume $\exists T_n$ s.t. $e \in \bar{\mathbb{T}}|T_n$.

We assume without loss of generality that $T_n \dot{\sim} T_k$. Thus, there is a view event e_v^n and possibly a routine update event e_u^n in $\bar{\mathbb{T}}|T_n$. From minimalism and Lemma 7: $e_v^n <_{\bar{\mathbb{T}}} e_u^n$. Also, since $T_n \dot{\sim} T_k$, then $T_k \dot{<}_{\bar{\mathbb{T}}}^x T_n$, so from Corollary 2, $\text{pv}_k(x) < \text{pv}_n(x)$.

If T_l executes e_a^l , then from Lemma 18, $\exists e_u^l = \circ s_l(x) \square$ in $\bar{\mathbb{T}}|T_l$ s.t. $e_u^k <_{\bar{\mathbb{T}}} e_a^l$. Hence either $e_u^l <_{\bar{\mathbb{T}}} e_u^k$ or $e_u^k <_{\bar{\mathbb{T}}} e_u^l$. So either $T_k \dot{<}_{\bar{\mathbb{T}}}^x T_l \dot{<}_{\bar{\mathbb{T}}}^x T_n$ or $T_k \dot{<}_{\bar{\mathbb{T}}}^x T_n \dot{<}_{\bar{\mathbb{T}}}^x T_l$. Thus, from Corollary 2, either $\text{pv}_k(x) < \text{pv}_l(x) < \text{pv}_n(x)$ or $\text{pv}_k(x) < \text{pv}_n(x) < \text{pv}_l(x)$.

If $\text{pv}_k(x) < \text{pv}_l(x) < \text{pv}_n(x)$, then either $e_a^l <_{\bar{\mathbb{T}}} e_v^n$ or $e_v^n <_{\bar{\mathbb{T}}} e_a^l$.

If $e_a^l <_{\bar{\mathbb{T}}} e_v^n$, then since e_a^l sets x to v' s.t. $v' \neq v''$ for any v'' s.t. $\exists \circ s_l(x)v'' \in \bar{\mathbb{T}}|T_l$ prior to the occurrence of e_v^n . Thus when T_n subsequently executes **checkpoint** it issues $e_v^n = g_n(x)v'$, and since $v' \neq v''$, this contradicts that $T_n \dot{\sim} T_k$.

If $e_v^n <_{\bar{\mathbb{T}}} e_a^l$, then since $T_l \dot{<}_{\bar{\mathbb{T}}}^x T_n$ then from version order $\text{pv}_l(x) < \text{pv}_n(x)$, so from Lemma 9, $\text{rv}_l(x) < \text{rv}_n(x)$. In order for T_n to issue e_a , it must execute **abort** and satisfy the condition line 87. This means that line 91 is executed, so $\text{cv}(x) = \text{rv}_l(x)$.

If subsequently T_l issues e_u^n then it either executes **write_buffer** or **commit**. Issuing an update event at line 56 or line 113 is dominated by checking

whether $\text{cv}(x) = \text{rv}_n(x)$ (for all variables) at line 54 or line 110, respectively. If the condition is failed, the transaction aborts instead. From Lemma 9, $\text{rv}_l(x) < \text{rv}_n(x)$, so if $\text{cv}(x) = \text{rv}_l(x)$, then $\text{cv}(x) \neq \text{rv}_n(x)$. Hence, T_i will **abort** rather than issue an update event. Since during **abort** only a recovery event may be issued, and only if $\text{rv}_n(x) = \text{cv}(x)$, then, similarly, no recovery event is issued. Hence T_n cannot issue events on x following e_a^l . Since each occurrence of a routine update event or a view event checks $\forall y, \text{cv}(y) = \text{rv}_n(y)$, then no other such event in T_n can follow e_a^l . This is a contradiction.

The execution of a recovery event on x by T_l is dominated by line 84, which cannot be passed until $\text{pv}_l(x) - 1 = \text{ltv}(x)$. Any transaction T_n sets $\text{ltv}(x)$ to $\text{pv}_o(x)$ as a last action during **commit** (line 80) or **abort** (line 95). Hence T_l cannot proceed to abort until T_o finishes committing or aborting. Since T_o cannot execute line 80 or line 95 if line 68 or line 84 was passed, then T_o cannot proceed to commit or abort until some other T_m s.t. $\text{pv}_o(x) - 1 = \text{pv}_m(x)$ committed or aborted. Hence T_l cannot execute a recovery event until any T_m s.t. $\text{pv}_m(x) < \text{pv}_l(x)$ committed or aborted.

If $\text{pv}_k(x) < \text{pv}_n(x) < \text{pv}_l(x)$, from version order $\text{pv}_n(x) < \text{pv}_l(x)$, so T_l executes **abort** only after T_k returns from **abort** or **commit**. Hence, since $e_u^n <_{\mathbb{T}} \text{res}_n(A_n)$ or $e_u^n <_{\mathbb{T}} \text{res}_n(C_n)$, either $\text{res}_n(A_n) \in \mathbb{T}|T_n$ or $\text{res}_n(C_n) \in \mathbb{T}|T_n$, and since $\text{res}_n(C_n) <_{\mathbb{T}} e_a^l$ or $\text{res}_n(A_n) <_{\mathbb{T}} e_a^l$, then $e_u^n <_{\mathbb{T}} e_a^l$. This contradicts that $e_a^l <_{\mathbb{T}} e_u^n$.

Thus, regardless of whether there is a contradiction. Therefore, T_l cannot issue such e_a^l between e_u^k and another event in $\mathbb{T}|T_n$. By extension, it cannot issue e_a^l between e_u^k and another event in $\mathbb{T}|T_m$ for any $T_m \in \xi(\mathbb{T}, T_i, T_j)$.

□

Lemma 33 (Chain Self-containment). *Given \mathbb{T} , and any transactions $T_i, T_j \in \mathbb{T}$, s.t. $\exists \xi(\mathbb{T}, T_i, T_j)$, $\xi(\mathbb{T}, T_i, T_j)$ is self-contained.*

Proof. Given transaction T_q s.t. $\exists e_v^q = g_q(x)v^q \in \mathbb{T}|T_q$, assuming that there are any update events on x in \mathbb{T} prior to e_v^q , then $\exists e_u^r = \circ s_r(x)v^q \in \mathbb{T}|T_r$ where $e_u^r < e_v^q$ or $\exists e_a^r = \blacklozenge s_r(x)v^q \in \mathbb{T}|T_r$ where $e_a^r < e_v^q$ for some T_r . In addition, from Lemma 7, $\exists e_v^r = g_r(x)v^r \in \mathbb{T}|T_r$ s.t. $e_v^r <_{\mathbb{T}} e_u^r$ and $e_v^r <_{\mathbb{T}} e_a^r$ (as applicable).

Then, similarly, assuming that there are any update events on x in \mathbb{T} prior to e_v^r , then $\exists e_u^s = \circ s_s(x)v^r \in \mathbb{T}|T_s$ where $e_u^s < e_v^r$ or $\exists e_a^s = \blacklozenge s_s(x)v^r \in \mathbb{T}|T_s$ where $e_a^s < e_v^r$. And by analogy to T_r , from Lemma 7, $\exists e_v^r = g_r(x)v^r$ s.t. $e_v^r <_{\mathbb{T}} e_u^s$ and $e_v^r <_{\mathbb{T}} e_a^s$ (as applicable).

It is then clear that as long as there are update events on x preceding a view event in some transaction, another transaction exists that both views and updates x before that view event.

Thus, given T_k and $T_l \in \xi(\mathbb{T}, T_i, T_j)$ such that $k \neq l$ and $\exists e_u^k = \circ s_k(x)v \in \mathbb{T}|T$ $\exists e_v^l = g_l(x)v' \in \mathbb{T}|T$ and $e_u^k <_{\mathbb{T}} e_v^l$, there is a sequence of transactions \mathcal{S} s.t.:

1. the first transaction is T_k ,
2. the last transaction is T_l , and

3. given some transaction $T_m \in \mathcal{S}$, where $m \neq k$, T_m is preceded in \mathcal{S} by some transaction T_n , s.t. for $e_v^m = g_m(x)v^m \in \bar{\mathbb{T}}|T_m$, $\exists e_u^n = \circ s_n(x)v^m \in \bar{\mathbb{T}}|T_n$ where $e_u^n < e_v^m$ or $\exists e_a^n = \blacklozenge s_n(x)v^m \in \bar{\mathbb{T}}|T_n$ where $e_a^n < e_v^m$.

Given such \mathcal{S} , given some T_m , $m \neq k$, there is some T_n that precedes T_m in \mathcal{S} .

If for for $e_v^m = g_m(x)v^m \in \bar{\mathbb{T}}|T_m$, $\exists e_u^n = \circ s_n(x)v^m \in \bar{\mathbb{T}}|T_n$ where $e_u^n < e_v^m$, then $T_m \dot{\leftrightarrow} T_n$.

If, on the other hand, for $e_v^m = g_m(x)v^m \in \bar{\mathbb{T}}|T_m$, or $\exists e_a^n = \blacklozenge s_n(x)v^m \in \bar{\mathbb{T}}|T_n$ where $e_a^n < e_v^m$, then from chain isolation there cannot be a recovery event $e_a^k = \blacklozenge s_k(x)v^m$ s.t. $e_a^k <_{\bar{\mathbb{T}}} e_v^l$, so it follows that $k \neq n$. Since e_a^n is conservative, $\exists e_v^n = g_n(x)v^m \in \bar{\mathbb{T}}|T_n$.

Since $k \neq n$ and $T_n \in \mathcal{S}$, then there is some T_o preceding T_n in \mathcal{S} . Then:

- a) If $o = k$, then $T_m \dot{\leftrightarrow} T_k$.
- b) If $o \neq k$ and $\exists e_u^n = \circ s_o(x)v^m \in \bar{\mathbb{T}}|T_o$ where $e_u^n < e_v^n$ then $T_m \dot{\leftrightarrow} T_o$.
- c) If $o \neq k$ and $\exists e_a^n = \blacklozenge s_o(x)v^m \in \bar{\mathbb{T}}|T_o$ where $e_a^n < e_v^n$ then by analogy, either case a), b) or c) applies to T_o as it does to T_n . So, by analogy, either a) $T_m \dot{\leftrightarrow} T_k$, b) $T_m \dot{\leftrightarrow} T_o$, or c) there is another preceding transaction in \mathcal{S} , etc.

Note, however, that since \mathcal{S} is finite, and e_a^k cannot precede e_v^l in $\bar{\mathbb{T}}$, then eventually for some such preceding $T_q \in \mathcal{S}$ case a) or b) and not c) will apply. Thus, there will be some $T_q \in \mathcal{S}$ s.t. $T_m \dot{\leftrightarrow} T_q$ (where either $q = k$ or $q \neq k$).

Therefore, $\forall T_m \in \mathcal{S}$ s.t. $m \neq k$, $\exists T_n \in \mathcal{S}$ s.t. $T_m \dot{\leftrightarrow} T_n$.

In addition, for each such pair T_m, T_n , there is therefore $\xi(\bar{\mathbb{T}}, T_n, T_m)$. Furthermore, if $T_m \dot{\leftrightarrow} T_n$ and for some other T_o , $T_n \dot{\leftrightarrow} T_o$, then there is $\xi(\bar{\mathbb{T}}, T_o, T_m)$. Thus, there is also $\xi(\bar{\mathbb{T}}, T_k, T_l)$, such that if $T_l \dot{\leftrightarrow} T_m$ and $T_m \in \mathcal{S}$, then $T_m \in \xi(\bar{\mathbb{T}}, T_k, T_l)$. Since $T_k, T_l \in \xi(\bar{\mathbb{T}}, T_i, T_j)$, then if $T_m \in \xi(\bar{\mathbb{T}}, T_k, T_l)$, $T_m \in \xi(\bar{\mathbb{T}}, T_i, T_j)$.

If $\mathcal{S} = T_k \cdot T_l$ then trivially $v = v'$.

Otherwise, since $T_l \in \mathcal{S}$ and $l \neq k$, then $\exists T_m \in \mathcal{S}$ s.t. $T_l \dot{\leftrightarrow} T_m$, so $\exists e_u^m = \circ s_m(x)v' \in \bar{\mathbb{T}}|T_m$ for some $T_m \in \xi(\bar{\mathbb{T}}, T_i, T_j)$ s.t. T_m precedes T_l and follows T_k in $\xi(\bar{\mathbb{T}}, T_i, T_j)$ and $e_u^k <_{\bar{\mathbb{T}}} e_u^m <_{\bar{\mathbb{T}}} e_v^l$. \square

Lemma 34 (Tree Chain Consistency). *Trace $\bar{\mathbb{T}}$ is chain consistent.*

Proof. From Lemma 32 and Lemma 33. \square

Lemma 35 (Trace Harmony). *Trace $\bar{\mathbb{T}}$ is harmonious.*

Proof. Trace $\bar{\mathbb{T}}$ satisfies all of the following: a) minimalism from Lemma 5 b) consonance from Lemma 22, c) obbligato from Lemma 26, d) coherence, commit accord, abort accord, and abort coda from Lemmas 31, 29, 28, and 30, e) isolation from Lemma 10, f) decisiveness from Lemma 27, g) chain consistency from Lemma 34, h) unique writes (assumed). \square

Corollary 4. *History $H = \text{hist}(\bar{\mathbb{T}})$ is last-use opaque. In consequence OptSVA is last-use opaque.*

7 Conclusions

The paper presents OptSVA, a highly optimized pessimistic TM that uses a number of techniques to improve the length of interleavings produced by conflicting transactions without losing the guarantees of the algorithm it was based on. Some of these techniques are straightforward, like the read-write distinction and buffering, but were sorely lacking in versioning algorithms thus far. More importantly, using dedicated executor threads to do transactions' waiting for them, so they can perform other computations in the mean time, is a novel technique in the context of TM. The agglomeration of these techniques creates a new algorithm that performs well in comparison to other versioning algorithms, both in theory, as well as in practice. On the other hand, OptSVA maintains the advantages of pessimistic TM with regards to irrevocable operations, since it only aborts when an abort is manually invoked.

In addition to the algorithm itself, the paper also presents trace harmony, a proof technique that can be used for OptSVA and other buffered algorithms that concentrate on memory accesses over transactional API operations to demonstrate last-use opacity. Moreover, the proof technique can be trivially modified to demonstrate opacity (and related properties). This requires applying the tests that we apply to events in committed transactions to all other transactions uniformly.

Future work on OptSVA includes a comprehensive evaluation comparing it against state-of-the-art TM algorithms. However, even though OptSVA can be implemented successfully as a multicore system, we consider its intended application to be in distributed TM, and such an implementation is out of scope of this paper, even though, the results of the evaluation of such a system are promising.

Acknowledgments The project was funded from National Science Centre funds granted by decision No. DEC-2012/06/M/ST6/00463.

References

- [1] Y. Afek, A. Matveev, and N. Shavit. Pessimistic Software Lock-Elision. In *Proceedings of DISC'12: the 26th International Symposium on Distributed Computing*, pages 297–311, Oct. 2012.
- [2] H. Attiya and E. Hillel. Single-version STMs Can Be Multi-version Permissive. In *Proceedings of ICDCN'11: the 12th International Conference on Distributed Computing and Networking*, number 6522 in Lecture Notes in Computer Science, pages 83–94, Jan. 2011.
- [3] H. Avni, S. Dolev, P. Fatourou, and E. Kosmas. Abort free semantic tm by dependency aware scheduling of transactional instructions. In *Proceedings of NETYS'14*.
- [4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

- [5] A. Bieniusa, A. Middelkoop, and P. Thiemann. Brief Announcement: Actions in the Twilight—Concurrent Irrevocable Transactions and Inconsistency Repair. In *Proceedings of PODC'10: the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2010.
- [6] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of DISC'06: the 20th International Symposium on Distributed Computing*, Sept. 2006.
- [7] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25:769–799, Sept. 2013.
- [8] D. Dziuina, P. Fatourou, and E. Kanellou. Consistency for transactional memory computing. *Bulletin of the EATCS*, 113, 2014.
- [9] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Morgan & Claypool, 2010.
- [10] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of PPOPP'05: the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.
- [11] M. Herlihy, V. Luchangco, M. Moir, and I. W. N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of PODC'03: the 22nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.
- [12] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of ISCA'93: the 20th International Symposium on Computer Architecture*, pages 289–300, May 1993.
- [13] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal tm characteristics. In *Proceedings of IISWC'10: the IEEE International Symposium on Workload Characterization*, 2010.
- [14] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. C. Kirkham, and I. Watson. DiSTM: A Software Transactional Memory Framework for Clusters. In *Proceedings of ICPP'08: the 37th IEEE International Conference on Parallel Processing*, Sept. 2008.
- [15] M. Lesani and J. Palsberg. Decomposing opacity. In *Proceedings of DISC'14: the 28th International Symposium on Distributed Computing*, 2014.
- [16] A. Matveev and N. Shavit. Towards a Fully Pessimistic STM Model. In *Proceedings of TRANSACT '12: the 7th ACM SIGPLAN Workshop on Transactional Computing*, number 7437 in Lecture Notes in Computer Science, pages 192–206, Aug. 2012.
- [17] C. H. Papadimitrou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, 1979.

- [18] D. Perelman, R. Fan, and I. Keidar. On Maintaining Multiple Versions in STM. In *Proceedings of PODC'10: the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2010.
- [19] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing Conflicting Transactions in an STM. In *Proceedings of PPOPP'09: the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2009.
- [20] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of PODC'95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 204–213, Aug. 1995.
- [21] K. Siek and P. T. Wojciechowski. A Formal Design of a Tool for Static Analysis of Upper Bounds on Object Calls in Java. In *Proceedings of FMICS'12: the 17th International Workshop on Formal Methods for Industrial Critical Systems*, number 7437 in Lecture Notes in Computer Science, pages 192–206, Aug. 2012.
- [22] K. Siek and P. T. Wojciechowski. Brief announcement: Towards a Fully-Articulated Pessimistic Distributed Transactional Memory. In *Proceedings of SPAA'13: the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 111–114, July 2013.
- [23] K. Siek and P. T. Wojciechowski. Brief announcement: Relaxing opacity in pessimistic transactional memory. In *Proceedings of DISC'14: the 28th International Symposium on Distributed Computing*, 2014.
- [24] K. Siek and P. T. Wojciechowski. Atomic RMI: A Distributed Transactional Memory Framework. *International Journal of Parallel Programming*, 2015.
- [25] K. Siek and P. T. Wojciechowski. Last-use opacity: A strong safety property for transactional memory with early release support, June 2015. preprint, <http://arxiv.org/abs/1506.06275>.
- [26] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Proceedings of WTW'06: the Workshop on Transactional Memory Workloads*, June 2006.
- [27] A. Turcu, B. Ravindran, and R. Palmieri. HyFlow2: A High Performance Distributed Transactional Memory Framework in Scala. In *Proceedings of PPPJ'13: the 10th International Conference on Principles and Practices of Programming on JAVA platform: virtual machines, languages, and tools*, Sept. 2013.
- [28] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, 2002.
- [29] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of SPAA'08: the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.

- [30] P. T. Wojciechowski. Isolation-only Transactions by Typing and Versioning. In *Proceedings of PPDP'05: the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, July 2005.
- [31] P. T. Wojciechowski. *Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems*. Poznań University of Technology Press, 2007.

A Last-use Opacity from Harmony

A.1 Last-use Opacity

Given program \mathbb{P} and a set of processes Π executing \mathbb{P} , since different interleavings of Π cause an execution $\mathcal{E}(\mathbb{P}, \Pi)$ to produce different histories, then let $\mathbb{H}^{\mathbb{P}, \Pi}$ be the set of all possible histories that can be produced by $\mathcal{E}(\mathbb{P}, \Pi)$, i.e., $\mathbb{H}^{\mathbb{P}, \Pi}$ is the largest possible set s.t. $\mathbb{H}^{\mathbb{P}, \Pi} = \{H \mid H \models \mathcal{E}(\mathbb{P}, \Pi)\}$.

Definition 26 (Closing Write Invocation [25]). *Given a program \mathbb{P} , a set of processes Π executing \mathbb{P} and a history H s.t. $H \models \mathcal{E}(\mathbb{P}, \Pi)$, i.e. $H \in \mathbb{H}^{\mathbb{P}, \Pi}$, an invocation $\text{inv}_i(w(x)v)$ is the closing write invocation on some variable x by transaction T_i in H , if for any history $H' \in \mathbb{H}^{\mathbb{P}, \Pi}$ for which H is a prefix (i.e., $H' = H \cdot R$) there is no operation invocation $\text{inv}_i(w(x)u)$ s.t. $\text{inv}_i(w(x)v)$ precedes $\text{inv}_i(w(x)u)$ in $H'|T_i$.*

Definition 27 (Closing Write [25]). *Given a program \mathbb{P} , a set of processes Π executing \mathbb{P} and a history H s.t. $H \models \mathcal{E}(\mathbb{P}, \Pi)$, an operation execution is the closing write on some variable x by transaction T_i in H if it comprises of an invocation and a response other than A_i , and the invocation is the closing write invocation on x by T_i in H .*

Definition 28 (Transaction Decided on x [25]). *Given a program \mathbb{P} , a set of processes Π and a history H s.t. $H \models \mathcal{E}(\mathbb{P}, \Pi)$, we say transaction $T_i \in H$ decided on variable x in H iff $H|T_i$ contains a complete write operation execution $w_i(x)v \rightarrow \text{ok}_i$ that is the closing write on x .*

Given some history H , let $\hat{\mathbb{T}}^H$ be a set of transactions s.t. $T_i \in \hat{\mathbb{T}}^H$ iff there is some variable x s.t. T_i decided on x in H .

Given any $T_i \in H$, a *decided transaction subhistory*, denoted $H\hat{|}T_i$, is the longest subsequence of $H|T_i$ s.t.:

- a) $H\hat{|}T_i$ contains $\text{start}_i \rightarrow u$, and
- b) for any variable x , if T_i decided on x in H , then $H\hat{|}T_i$ contains $(H|T_i)|x$.

In addition, a *decided transaction subhistory completion*, denoted $H\hat{|}^\circ T_i$, is a sequence s.t. $H\hat{|}^\circ T_i = H\hat{|}T_i \cdot [\text{try}C_i \rightarrow C_i]$.

Given a sequential history S s.t. $S \equiv H$, $LVis(S, T_i)$ is the longest subhistory of S , s.t. for each $T_j \in S$:

- a) $S|T_j \subseteq LVis(S, T_i)$ if $i = j$ or T_j is committed in S and $T_j <_S T_i$, or
- b) $S\hat{|}^\circ T_j \subseteq LVis(S, T_i)$ only if T_j is not committed in S but $T_j \in \hat{\mathbb{T}}^H$ and $T_j <_S T_i$ and not $T_j <_H T_i$.

Given a sequential history S and a transaction $T_i \in S$, we then say that transaction T_i is *last-use legal* in S if $LVis(S, T_i)$ is legal.

Definition 29 (Final-state Last-use Opacity [25]). *A finite history H is final-state last-use opaque if, and only if, there exists a sequential history S equivalent to any completion of H s.t.,*

- a) S preserves the real-time order of H ,

- b) every transaction in S that is committed in S is legal in S ,
- c) every transaction in S that is not committed in S is last-use legal in S .

Definition 30 (Last-use Opacity [25]). *A history H is last-use opaque if, and only if, every finite prefix of H is final-state last-use opaque.*

A.2 Composition Rules

Given trace \mathbb{T} and a history $H = \text{hist}(\mathbb{T})$, let $\hat{C} = \text{Compl}(H)$ be a completion of H s.t. for every $T_i \in H$, if T_i is live or commit-pending in H , then T_i is aborted in H_C . Let \hat{T}_i such a transaction in \hat{C} that corresponds to a completion of T_i in \hat{C} .

Definition 31 (Equivalent Sequential History Construction). *Let \hat{S}_H be a sequential history s.t. $\hat{S}_H \equiv H_C$ and, given two transactions $T_i, T_j \in \hat{C}$:*

1. if $T_i <_{\mathbb{T}} T_j$, then $T_i <_{\hat{S}_H} T_j$,
2. otherwise, if $T_i <_{\mathbb{T}} T_j$ for any variable x , then $T_i <_{\hat{S}_H} T_j$,
3. otherwise, if $\exists \text{op}_j = w_j(x) \square \rightarrow \text{ok}_j \in \mathbb{T}|T_j$ and $\exists e^i = g_i(x) \square \in \mathbb{T}|T_i$ or $e^i = \text{os}_i(x) \square \in \mathbb{T}|T_i$, then $T_i <_{\hat{S}_H} T_j$.

Definition 32 (Last-use Visible History Construction). *Given transactions T_i and T_j in \mathbb{T} :*

1. if T_j is committed in \mathbb{T} , then \hat{T}_j is included in $L\text{Vis}(\hat{S}_H, T_i)$ as a whole, otherwise
2. if T_j is aborted in \mathbb{T} and $T_j <_{\mathbb{T}} T_i$, \hat{T}_j is not included in $L\text{Vis}(\hat{S}_H, T_i)$ at all, otherwise
3. if there exists $\xi(\mathbb{T}, T_j, T_i)$, then $\hat{S}_H \circ \hat{T}_j$ is included in $L\text{Vis}(\hat{S}_H, T_i)$, otherwise
4. T_j is not included in $L\text{Vis}(\hat{S}_H, T_i)$ at all.

A.3 Auxilia

Lemma 36. *Let there be a consonant, isolation-ordered, trace \mathbb{T} in obligato and $H = \text{hist}(\mathbb{T})$ from which \hat{S}_H is generated, and $T_i, T_j \in \mathbb{T}$. Given any non-local $\text{op}_i = r_i(x) \rightarrow v \in \mathbb{T}|T_i$ s.t. $\exists e_i = g_i(x)v \in \mathbb{T}|T_i$ and $\text{op}_i \leftarrow e_i$ and given any non-local $\text{op}_j = w_j(x)v \rightarrow \text{ok}_j \in \mathbb{T}|T_j$ s.t. $\exists e_j = \text{os}_i(x)v \in \mathbb{T}|T_j$ and $\text{op}_j \rightsquigarrow e_j$, and $e_j <_{\mathbb{T}} e_i$, then $\nexists T_k \in \mathbb{T}$ s.t. $T_k \text{op}_k = w_j(x)v' \rightarrow \text{ok}_j$ s.t. $\text{op}_i <_{\hat{S}_H} \text{op}_k <_{\hat{S}_H} \text{op}_j$ and T_k is either committed or decided on x in trace \mathbb{T} .*

Proof. Assume for the sake of contradiction that such T_k exists in \mathbb{T} . Since both op_i and op_j are non-local, then $i \neq j \neq k$.

If T_k is committed, then, from the definition of commit write obligato, $\exists e_k = \text{os}_k(x)v' \in \mathbb{T}|T_k$ if $\text{inv}_k(w_k(x)v')$ is the invocation event of op_k then $\text{inv}_k(w_k(x)v') <_{\mathbb{T}} e_k <_{\mathbb{T}} \text{res}_k(C_k)$.

If T_k is decided on x in \mathbb{T} , then, from the definition of closing write obligato, $\exists e_k = \circ s_k(x)v' \in \mathbb{T}|T_k$ s.t. if $inv_k(w_k(x)v')$ is the invocation event of op_k then $inv_k(w_k(x)v') <_{\mathbb{T}} e_k <_{\mathbb{T}} e_i$.

Thus, in either of the above cases, $e_j <_{\mathbb{T}} e_k$ and either $e_k < e_i$ or $e_i < e_k$. If then $e_k < e_i$, it is not true that $e_j <_{\mathbb{T}} e_i$, which is a contradiction. Alternatively, if $e_i < e_k$, then, since \mathbb{T} is isolation-ordered, $T_i \dot{<}^x_{\mathbb{T}} T_k$, which implies that $T_i <_{\hat{S}_H} T_k$. In this case, $op_i <_{\hat{S}_H} op_k$, which is a contradiction.

Therefore, there can be no such T_k , which satisfies the lemma. \square

Lemma 37. *Given a consonant trace \mathbb{T} , and $T_i \in \mathbb{T}$, if T_j is the first element of $\psi_{\mathbb{T}}(T_j, x)$, then $\exists e_v = g_j(x)v \in \mathbb{T}|T_j$ that is initial and non-local, and either*

- a) $v = 0$ and $\nexists T_k \in \mathbb{T}$ s.t. $e_u = s_k(x)v \in \mathbb{T}|T_k$ and $e_u <_{\mathbb{T}} e_v$,
- b) $v \neq 0$ and $\exists T_k \in \mathbb{T}$ s.t. $e_u = s_k(x)v \in \mathbb{T}|T_k$ and $e_u <_{\mathbb{T}} e_v$.

Proof. Since T_j is in $\psi_{\mathbb{T}}(T_i, x)$ then by definition, either $k = i$ or $e_a = \diamond s_j(x)v \in \mathbb{T}|T_j$. In either case $e_v = g_j(x)v \in \mathbb{T}|T_j$ s.t. e_v is initial and non-local (in the former case by definition of $\psi_{\mathbb{T}}(T_i, x)$ and in the latter by definition of recovery update consonance).

Since e_v is consonant and non-local, then either:

- a) $v = 0$ and $\nexists T_k \in \mathbb{T}$ s.t. $e_u s_k(x)v' \in \mathbb{T}|T_k$ $e_u <_{\mathbb{T}} e_r$,
- b) $v \neq 0$ and $\exists T_k \in \mathbb{T}$ s.t. $e_u = \circ s_j(x)v \in \mathbb{T}|T_k$, $i \neq k$, $e_u <_{\mathbb{T}} e_r$, e_u is consonant, and e_u is the ultimate routine update on x in $\mathbb{T}|T_k$, or
- c) $\exists e_u \in \mathbb{T}$ s.t. $e_u = \diamond s_j(x)v$ for some tr_k , $j \neq k$, $e_u <_{\mathbb{T}} e_r$, e_u is a consonant recovery event, and is the ultimate update on x in $\mathbb{T}|T_k$.

In the latter-most case, if such e_u exists in T_k then, $T_k \in \psi_{\mathbb{T}}(T_j, x)$ so that T_k preceded T_j in $\psi_{\mathbb{T}}(T_j, x)$. Thus, T_k would precede T_j in $\psi_{\mathbb{T}}(T_i, x)$, and therefore T_j is not the first element of $\psi_{\mathbb{T}}(T_i, x)$. Thus, the latter-most case is impossible. \square

Lemma 38. *Given a consonant trace \mathbb{T} , and $T_i \in \mathbb{T}$, $\forall T_j \in \psi_{\mathbb{T}}(T_i, x)$ ($i \neq j$), T_j is aborted or live in \mathbb{T} .*

Proof. Since $i \neq j$ then $\forall T_j \in \mathbb{T}$, $\exists e_a = \diamond s_j(x)v \in \mathbb{T}|T_j$. Since \mathbb{T} is consonant, then e_a is consonant, so e_a is dooming. Thus T_j is aborted or live in \mathbb{T} . \square

Lemma 39. *Given a consonant, abort abiding trace \mathbb{T} in obligato, and a pair of transaction $T_i, T_j \in \mathbb{T}$, and T_j is the first element in $\psi_{\mathbb{T}}(T_i, x)$, $\forall T_k \in \mathbb{T}$ if $T_j \dot{<}^x_{\mathbb{T}} T_k \dot{<}^x_{\mathbb{T}} T_i$ and $op_k w_k(x)v \rightarrow \in \mathbb{T}|T_k$ then T_k is aborted or live in \mathbb{T} .*

Proof. If $i = j$, then the lemma is vacuously true.

Since $T_j \dot{<}^x_{\mathbb{T}} T_k \dot{<}^x_{\mathbb{T}} T_i$, then $\exists e_u = \circ s_k(x)v' \in \mathbb{T}|T_k$ or $e_v = g_k(x)v' \in \mathbb{T}|T_k$. Hence, either e_u exists in $\mathbb{T}|T_k$ or it does not.

If e_u does not exist, then, from commit write obligato, T_k cannot commit in trace, so T_k is either live or aborted in \mathbb{T} .

If e_u exists, then, since $T_j \dot{<}^x_{\mathbb{T}} T_k \dot{<}^x_{\mathbb{T}} T_i$ and from the definition of $\psi_{\mathbb{T}}(T_i, x)$, there is some pair of transactions T_α and $T_\beta \in \mathbb{T}$ s.t. $T_\alpha, T_\beta \in \psi_{\mathbb{T}}(T_i, x)$ and T_α immediately precedes T_β in $\psi_{\mathbb{T}}(T_i, x)$ and $T_\alpha \dot{<}^x_{\mathbb{T}} T_k \dot{<}^x_{\mathbb{T}} T_\beta$. Therefore $\exists e_\alpha = \diamond s_\alpha(x)v_\alpha \in \mathbb{T}|T_\alpha$ and $e_\beta = g_\beta(x)v_\beta \in \mathbb{T}|T_\beta$ s.t. $e_\alpha <_{\mathbb{T}} e_\beta$. In addition, since e_α

is consonant, then it is needed, so $\exists e'_\alpha = \circ s_\alpha(x)v'_\alpha \in \mathbb{T}|T_\alpha$ s.t. $e'_\alpha <_{\mathbb{T}} e_\alpha$. Also, from definition of isolation order, $e'_\alpha <_{\mathbb{T}} e_u <_{\mathbb{T}} e_\beta$. Then, $e'_\alpha <_{\mathbb{T}} e_u <_\alpha e_\beta$. Therefore, from the definition of abort accord, T_k is either live or aborted in \mathbb{T} . \square

Lemma 40. *Given a consonant trace \mathbb{T} , and $T_i \in \mathbb{T}$, $\forall T_j, T_k \in \psi_{\mathbb{T}}(T_i, x)$ ($i \neq j$), if T_j precedes T_k in $\psi_{\mathbb{T}}(T_i, x)$ then $T_j \dot{<}^x_{\mathbb{T}} T_k$.*

Proof. Given $\psi_{\mathbb{T}}(T_i, x)$, from Lemma 38, $\forall T_k \in \psi_{\mathbb{T}}(T_i, x)$, T_k is aborted or live in \mathbb{T} . In addition, since for all $T_m \in \psi_{\mathbb{T}}(T_i, x)$ except the first, where $e_v^m = g_m(x)v \in \mathbb{T}|T_m$ there is some T_n that directly precedes T_m in $\psi_{\mathbb{T}}(T_i, x)$ and contains $e_a^n = \blacklozenge s_n(x)v$ s.t. $e_a^n <_{\mathbb{T}} e_v^m$. Since e_a^n is conservative, there is a preceding view $e_v^n = g_n(x)v$ s.t. $e_v^n <_{\mathbb{T}} e_a^n$. Thus $e_v^n <_{\mathbb{T}} e_v^m$, so $T_n \dot{<}^x_{\mathbb{T}} T_m$. \square

Corollary 5. *Given a consonant trace \mathbb{T} , and $T_i \in \mathbb{T}$, $\forall T_j \in \psi_{\mathbb{T}}(T_i, x)$ ($i \neq j$), $T_j \dot{<}^x_{\mathbb{T}} T_i$.*

Lemma 41. *Given T_i, T_j s.t. $\hat{T}_j \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_i)$, $\forall T_k$ if $\hat{T}_k \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_j)$, then $\hat{T}_k \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_i)$ and $LVis(\hat{S}_H, T_i)|T_k = LVis(\hat{S}_H, T_j)|T_k$*

Proof. If T_k is committed in \mathbb{T} and $\hat{T} \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_j)$ then $\hat{S}_H|\hat{T}_k \subseteq LVis(\hat{S}_H, T_j)$ and $\hat{T}_k <_{\hat{S}_H} \hat{T}_j$. Since $T_j \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_i)$, then $\hat{T}_j <_{\hat{S}_H} \hat{T}_i$. Since T_k is committed in \mathbb{T} and $\hat{T}_j <_{\hat{S}_H} \hat{T}_i$, then $\hat{S}_H|\hat{T}_k \subseteq LVis(\hat{S}_H, T_i)$.

If T_k is not committed in \mathbb{T} and $\hat{T} \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_j)$ then $\hat{S}_H|\hat{T}_k = LVis(\hat{S}_H, T_j, |)T_k$ and $\hat{T}_k <_{\hat{S}_H} \hat{T}_j$ and $T_k \not\prec_{\mathbb{T}} T_j$ and $\exists \xi(\mathbb{T}, T_k, T_j)$ (from Def. 32).

Since \hat{T}_k is not committed in \mathbb{T} , and since \mathbb{T} is commit abiding, then from Lemma 43, there cannot be $\xi(\mathbb{T}, T_k, T_j)$ s.t. T_j is committed. Thus T_j is not committed in \mathbb{T} . Thus, if $\hat{S}_H|\hat{T}_j$ then $\hat{T}_j <_{\hat{S}_H} \hat{T}_i$ and $T_j \not\prec_{\mathbb{T}} T_i$ and $\exists \xi(\mathbb{T}, T_j, T_i)$.

If $\exists \xi(\mathbb{T}, T_k, T_j)$ and $\exists \xi(\mathbb{T}, T_j, T_i)$ then $\exists \mathbb{T} T_k T_i$.

Either T_k aborts in \mathbb{T} (i.e. $res_k(A_k)$) or T_k is live in \mathbb{T} . In the latter case trivially $T_k \not\prec_{\mathbb{T}} T_i$. In the former case, from Lemma 44, also $T_k \not\prec_{\mathbb{T}} T_i$.

Since $\exists \xi(\mathbb{T}, T_k, T_i)$ and $\hat{T}_k <_{\hat{S}_H} \hat{T}_i$ and $T_k \not\prec_{\mathbb{T}} T_i$ then $\hat{S}_H|\hat{T}_k = LVis(\hat{S}_H, T_i, |)T_k$ (from Def. 32). \square

Lemma 42. *Given T_i, T_j s.t. $\hat{T}_j \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_i)$, $\forall T_k$ if $\hat{T}_k \not\stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_j)$ and $\exists \xi(\mathbb{T}, T_k, T_j)$ then $\hat{T}_k \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_i)$.*

Proof. If $T_k \not\stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_j)$ and $\hat{T}_k <_{\hat{S}_H} \hat{T}_j$ then \hat{T}_k is not committed in \mathbb{T} .

If $\hat{S}_H|\hat{T}_k \not\subseteq LVis(\hat{S}_H, T_j)$ and $\hat{T}_k <_{\hat{S}_H} \hat{T}_j$ then either $T_k <_{\mathbb{T}} T_j$ or $\nexists \xi(\mathbb{T}, T_i, T_j)$. The latter case contradicts the assumptions of the lemma, hence $T_k <_{\mathbb{T}} T_j$.

If $T_k <_{\mathbb{T}} T_j$, then $\exists r = res_k(A_k) \in \mathbb{T}|T_k$ s.t. for every event e in $\mathbb{T}|T_j$, $r <_{\mathbb{T}} e$. Since $\exists \xi(\mathbb{T}, T_j, T_i)$ then there is some view event e_v^i in $\mathbb{T}|T_i$ and some update event e_u^j in $\mathbb{T}|T_j$ s.t. $e_u^j <_{\mathbb{T}} e_v^i$. Therefore $r <_{\mathbb{T}} e_u^j <_{\mathbb{T}} e_v^i$.

Since no events can occur in $\mathbb{T}|T_k$ after v , then for all events in e in $\mathbb{T}|T_k$ apart from r , $e <_{\mathbb{T}} r$. So, for any $\xi(\mathbb{T}, T_k, T_i)$ for any update event $e_u^k = \circ s_k(x)v \in \mathbb{T}|T_k$, $e_u^k <_{\mathbb{T}} r <_{\mathbb{T}} e_v^i$.

From abort coda, $\exists e_a^l = \blacklozenge s_l(x)v'$ s.t. $e_u^k <_{\mathbb{T}} e_a^l <_{\mathbb{T}} r$, and, from conservatism and unique routine updates, $v \neq v'$. Thus, since $e_u^k <_{\mathbb{T}} e_a^l <_{\mathbb{T}} e_v^i$ and $v \neq v'$,

there cannot be such $\xi(\mathbb{T}, T_k, T_i)$ that satisfies chain isolation, and therefore $\nexists \xi(\mathbb{T}, T_k, T_i)$.

Therefore, from Lemma 32, $\hat{S}_H | \hat{T}_k \not\subseteq LVis(\hat{S}_H, T_i)$.

Thus, $\hat{S}_H | \hat{T}_i \not\subseteq LVis(\hat{S}_H, T_i)$. \square

Lemma 43. *Given $\xi(\mathbb{T}, T_i, T_j)$, if T_j is committed in \mathbb{T} , then $\forall T_k \in \xi(\mathbb{T}, T_i, T_j)$, T_k is committed in \mathbb{T} .*

Proof. Given a pair of transaction $T_l, T_m \in \mathbb{T}$ s.t. $T_m \dot{\leftarrow} T_l$, from commit accord, if T_m is committed in \mathbb{T} , then T_l is also committed in \mathbb{T} .

If $\xi(\mathbb{T}, T_i, T_j) = T_i \cdot T_j$, then since T_j is committed in \mathbb{T} , then so is T_i .

Since, $\xi(\mathbb{T}, T_i, T_j) = T_i \xi(\mathbb{T}, T_i, T_k) \cdot T_j$ and T_k is such that $T_j \dot{\leftarrow} T_k$, then since T_j is committed in \mathbb{T} , then so is T_k . This follows recursively for $\xi(\mathbb{T}, T_i, T_k)$.

Thus every transaction in $\xi(\mathbb{T}, T_i, T_j)$ is committed in \mathbb{T} . \square

Lemma 44. *Given $\xi(\mathbb{T}, T_i, T_j)$, if T_i aborts in \mathbb{T} , then $T_i \not\prec_{\mathbb{T}} T_j$.*

Proof. Assume for the sake of contradiction that $T_i \prec_{\mathbb{T}} T_j$.

Thus, there exists $T_k \in \xi(\mathbb{T}, T_i, T_j)$ s.t. $T_k \dot{\leftarrow} T_i$, so $\exists e_u^i = \circ s_i(x)v \in \mathbb{T} | T_i$ and $e_v^k = g_k(x)v \in \mathbb{T} | T_k$ and $e_u^i \prec_{\mathbb{T}} e_v^k$.

If T_i is aborted, then, from abort coda, $\exists e_a^l = \circ s_l(x)v' \in \mathbb{T}$ s.t. $e_u^i \prec_{\mathbb{T}} e_a^l \prec_{\mathbb{T}} res_i(A_i)$ and from unique routine updates $v \neq v'$.

Since T_j is in $\xi(\mathbb{T}, T_i, T_j)$, $\exists e_v^j = g_j(y)v''$ and since $T_i \prec_{\mathbb{T}} T_j$, then $res_i(A_i) \prec_{\mathbb{T}} e_v^j$. Thus, $e_u^i \prec_{\mathbb{T}} e_a^l \prec_{\mathbb{T}} e_v^j$.

This contradicts chain isolation, so it is not true that $T_i \prec_{\mathbb{T}} T_j$, so $T_i \not\prec_{\mathbb{T}} T_j$. \square

A.4 Main Lemmas

Let there be a harmonious trace \mathbb{T} and $H = hist(\mathbb{T})$ from which \hat{S}_H is generated. Let there be such $T_i \in \mathbb{T}$ that T_i is committed in \mathbb{T} . Then:

Lemma 45 (Unique Routine Updates). *If \mathbb{T} is consonant, and \mathbb{T} has unique writes, then given any $s_i(x)v$ and $s_j(x)v'$ s.t. $v \neq v'$.*

Proof. Since both events are consonant, then for $s_i(x)v$ there exists $op_i = w_i(x)v^i \rightarrow ok_i$ s.t. $v = v^i$, and for $s_j(x)v'$ there exists $op_j = w_j(x)v^j \rightarrow ok_j$ s.t. $v' = v^j$. Since \mathbb{T} has unique writes, then $v^i \neq v^j$, so $v \neq v'$. \square

Lemma 46 (Non-local Read Consistency). *For any $op_i \in \mathbb{T} | T_i$ s.t. $op_i = r_i(op_i) \rightarrow v$ and op_i is non-local, then either:*

- 1) $v \neq 0$ and $\exists op_j \in Vis(\hat{S}_H, \hat{T}_i)$ for some T_j s.t. $op_j = w_j(x)v \rightarrow ok_j$, $op_j \prec_{Vis(\hat{S}_H, \hat{T}_i)} op_i$, or
- 2) $v = 0$ and $\nexists op_j \in Vis(\hat{S}_H, \hat{T}_i)$ s.t. $op_j = w_j(x)v \rightarrow ok_j$ and $op_j \prec_{Vis(\hat{S}_H, \hat{T}_i)} op_i$.

Proof for Lemma 46. Since op_i is consonant and non-local, then $\exists e_v = g_i(x)v \in \mathbb{T}$, s.t. $op_i \leftarrow e_v$ and e_v is consonant. Then, from e_v 's consonance, either:

- a) $v = 0$ and $\nexists e_u = s_j(x)v' \in \mathbb{T}$ for some $T_j \in \mathbb{T}$ s.t. $e_u <_{\mathbb{T}} e_v$.

In which case, if $\nexists op_j w_j(x)v' \rightarrow ok_i \in \mathbb{T}|T_j$ s.t. $op_j <_{\mathbb{T}} g_i(x)v$, then, $\nexists T_j \in \mathbb{T}$ s.t. $T_j <_{\mathbb{T}} T_i$ and $op_j \in \mathbb{T}|T_j$. Thus, from construction of \hat{S}_H , $\nexists \hat{T}_j \in \mathbb{T}$ s.t. $\hat{T}_j <_{\hat{S}_H} \hat{T}_i$ and $op_j \in \hat{S}_H|\hat{T}_j$. Thus, from construction of $Vis(\hat{S}_H, \hat{T}_i)$, for any such T_j , $\hat{S}_H|\hat{T}_j \not\subseteq Vis(\hat{S}_H, \hat{T}_i)$, so for any such T_j , $w_j(x)v \rightarrow ok_j \notin Vis(\hat{S}_H, \hat{T}_i)$ and $v = 0$.

On the other hand, if $\exists op_j w_j(x)v' \rightarrow ok_i \in \mathbb{T}|T_j$ s.t. $op_j <_{\mathbb{T}} g_i(x)v$, then if T_i is committed in \mathbb{T} , then, from the definition of commit write obligato, $\circ s_j(x)v' \in \mathbb{T}|T_j$, which contradicts the assumption of case a). Thus, T_i is not committed in \mathbb{T} , so \hat{T}_i is not committed in \hat{S}_H , and therefore $\hat{S}_H|\hat{T}_i \not\subseteq Vis(\hat{S}_H, \hat{T}_i)$. Thus for any such T_j , $w_j(x)v' \rightarrow ok_j \notin Vis(\hat{S}_H, \hat{T}_i)$ and $v = 0$.

- b) $v \neq 0$ and $\exists e_u = \circ s_j(x)v \in \mathbb{T}$ for some $T_j \in \mathbb{T}$ s.t. $e_u <_{\mathbb{T}} e_v$ and e_u is consonant.

Since e_u is consonant, then $\exists op_j = w_j(x)v \rightarrow ok_j \in \mathbb{T}|T_j$ s.t. op_j is non-local and consonant, and $op_j < \circ s_j(x)v$. Thus, since $e_u <_{\mathbb{T}} e_v$, $T_j <_{\mathbb{T}}^x T_i$, then, by construction, $\hat{T}_j < \hat{S}_H \hat{T}_i$.

Since T_i is committed in \mathbb{T} and $T_i \dot{\sim} T_j$, and since \mathbb{T} is commit-abiding, then T_j must be committed in \mathbb{T} . Thus \hat{T}_j is also committed in \hat{S}_H . Thus, $\hat{S}_H|\hat{T}_j \subseteq Vis(\hat{S}_H, T_i)$, and therefore $op_j <_{Vis(\hat{S}_H, T_i)} op_i$. Then, from Lemma 36, $op_j <_{Vis(\hat{S}_H, T_i)} op_i$. Thus, $w_j(x)v \rightarrow ok_j <_{Vis(\hat{S}_H, \hat{T}_i)} op_i$ and $v \neq 0$.

- c) $\exists e_a = \diamond s_j(x)v \in \mathbb{T}$ for some $T_j \in \mathbb{T}$ s.t. $e_a <_{\mathbb{T}} e_v$ and e_u is consonant.

Given $\psi_{\mathbb{T}}(T_i, x)$, from Lemma 38, $\forall T_k \in \psi_{\mathbb{T}}(T_i, x)$, T_k is aborted or live in \mathbb{T} . So, by construction, \hat{T}_k is aborted in \hat{S}_H , and therefore excluded from $Vis(\hat{S}_H, T_i)$. Thus for any $T_k \in \psi_{\mathbb{T}}(T_i, x)$, $\forall op_k = w_k(x)v \rightarrow \in \mathbb{T}|T_k$, $op_k \notin Vis(\hat{S}_H, T_i)$.

Given $\psi_{\mathbb{T}}(T_i, x)$, from Lemma 37, $\exists e'_v = g_k(x)v \in \mathbb{T}|T_k$ s.t. T_k is the first element of $\psi_{\mathbb{T}}(T_i, x)$ that is initial and non-local, and either of the following is true:

- i) $v = 0$ and $\nexists T_l \in \mathbb{T}$ s.t. $e'_u = s_l(x)v \in \mathbb{T}|T_l$ and $e'_u <_{\mathbb{T}} e'_v$.

Then, either $\exists e'_u \in s_l(x)v \in \mathbb{T}|T_l$ and $e'_v <_{\mathbb{T}} e'_u$ or $\nexists e'_u = s_l(x)v \in \mathbb{T}|T_l$.

If $\exists e'_u \in s_l(x)v \in \mathbb{T}|T_l$ and $e'_v <_{\mathbb{T}} e'_u$, then from Lemma 37, $e'_u = \circ s_l(x)v'$. Thus, by definition of isolation order, $T_k <_{\mathbb{T}}^x T_l$. Thus, if $T_l <_{\mathbb{T}}^x T_j$, then, from Lemma 39, T_l is aborted or live in \mathbb{T} , so, by construction, \hat{T}_l is aborted in \hat{S}_H . Therefore $\hat{T}_l \not\subseteq Vis(\hat{S}_H, T_i)$, so for any write operation execution $op_l = w_l(x)v' \rightarrow ok_l$ in any such T_l , $op_l \notin Vis(\hat{S}_H, T_i)$ (and $v = 0$).

Alternatively, if $T_j <_{\mathbb{T}}^x T_l$, then since $e_a <_{\mathbb{T}} e_v$, then it is not possible that $e_a <_{\mathbb{T}} e'_u < e_v$. By corollary, from the definition of isolation order, it is not possible that $T_j <_{\mathbb{T}}^x T_l <_{\mathbb{T}}^x T_i$. Then, by construction, $\hat{T}_i <_{\hat{S}_H} \hat{T}_l$, so $\hat{T}_l \not\subseteq Vis(\hat{S}_H, T_i)$. Therefore, for any write operation execution $op_l = w_l(x)v' \rightarrow ok_l$ in any such T_l , $op_l \notin Vis(\hat{S}_H, T_i)$ (and $v = 0$).

On the other hand, if $\nexists e'_u \in s_l(x)v \in \mathbb{T}|T_l$, then either $\mathbb{T}|T_l$ contains some write operation $op_l = w_l(x)v' \rightarrow ok_l$ or it does not. If it does not, then vacuously, for any write operation execution $op_l = w_l(x)v' \rightarrow ok_l$ in any

such T_l , $op_l \notin \text{Vis}(\hat{S}_H, T_i)$ (and $v = 0$). On the other hand, if $op_l \in \mathbb{T}|T_l$, then from commit write obbligato, since $\#e'_u = \circ_{s_l}(x)v \in \mathbb{T}|T_l$, then T_l is not committed in \mathbb{T} . Thus, \hat{T}_l is aborted in \hat{S}_H and $\hat{T}_l \not\subseteq \text{Vis}(\hat{S}_H, T_i)$. Thus, for any write operation execution $op_l = w_l(x)v' \rightarrow ok_l$ in any such T_l , $op_l \notin \text{Vis}(\hat{S}_H, T_i)$ (and $v = 0$).

- ii) $v \neq 0$ and $\exists T_l \in \mathbb{T}$ s.t. $e'_u = \circ_{s_l}(x)v \in \mathbb{T}|T_l$ and $e'_u <_{\mathbb{T}} e'_v$.

Since \mathbb{T} is consonant, then e'_u is consonant, so $\exists op_l = w_l(x)v \rightarrow ok_l$ s.t. $op_l \rightsquigarrow e'_u$.

Since for all $T_m \in \psi_{\mathbb{T}}(T_i, x)$, $e'_v = g_m(x)v''$ there is some T_n that directly precedes T_m in $\psi_{\mathbb{T}}(T_i, x)$ and contains $e''_a = \diamond_{s_n}(x)v''$ s.t. $e''_a <_{\mathbb{T}} e'_v$. Since e''_a is conservative, there is a preceding view $e'''_v = g_n(x)v''$ s.t. $e'''_v <_{\mathbb{T}} e''_a$. Thus $e'''_v <_{\mathbb{T}} e'_v$, so $T_n \dot{<}^x_{\mathbb{T}} T_m$. Therefore, $T_k \dot{<}^x_{\mathbb{T}} T_i$, and, by extension, since $e'_u <_{\mathbb{T}} e'_v$, $T_l \dot{<}^x_{\mathbb{T}} T_k$, then $T_l \dot{<}^x_{\mathbb{T}} T_i$. Since T_i is committed in \mathbb{T} , then since $T_l \dot{<}^x_{\mathbb{T}} T_i$, then, from commit coherence, T_l is either committed or aborted in T_j .

Transaction T_l cannot be aborted in \mathbb{T} , as follows. Let us assume by contradiction that T_l is aborted (i.e. $\exists r_a = \text{res}_l(A_l) \in \mathbb{T}|T_l$). Then, since \mathbb{T} has coda, then for some T_n , $\exists e'''_a = \diamond_{s_n}(x)v'''$ s.t. $e'_u <_{\mathbb{T}} e'''_a <_{\mathbb{T}} r$. Since e'_a is consonant, then since it is clean and $T_l \dot{<}^x_{\mathbb{T}} T_k$, then there is no recovery event following e'_v and preceding e'_a . In addition from commit coherence, T_l must abort before T_i commits, so, by extension e'''_a must precede $\text{res}_i(C_i)$ in \mathbb{T} . Thus either $e'''_a <_{\mathbb{T}} e'_v$ or $e'_a <_{\mathbb{T}} e'''_a <_{\mathbb{T}} r_a$. In the former case, if $e'''_a <_{\mathbb{T}} e'_v$, then this contradicts that e'''_a is consonant (ending), and if $e'_a <_{\mathbb{T}} e'''_a <_{\mathbb{T}} e'_v$, it contradicts that $e'_u <_{\mathbb{T}} e'_v$. On the other hand, if $e'_a <_{\mathbb{T}} e'''_a <_{\mathbb{T}} r_a$, then one of three scenarios is possible. If for some $T_m \in \psi_{\mathbb{T}}(T_i, x)$, s.t. $i \neq m$ and $g_m(x) \square <_{\mathbb{T}} e'''_a <_{\mathbb{T}} \diamond_{s_m}(x) \square$, then this contradicts that $\diamond_{s_m}(x) \square$ is clean. Alternatively, if for a pair $T_m, T_n \in \psi_{\mathbb{T}}(T_i, x)$, s.t. T_m directly precedes T_n in $\psi_{\mathbb{T}}(T_i, x)$, and $\diamond_{s_m}(x) \square <_{\mathbb{T}} e'''_a <_{\mathbb{T}} g_n(x) \square$, then this contradicts that $\diamond_{s_m}(x) \square \leq_{\mathbb{T}} g_n(x) \square$. Finally, if $e_v < e'''_a$, then this violates abort coda of \mathbb{T} (case b), and is also a contradiction. Thus, there cannot be such e'''_a , and therefore T_l cannot be aborted in \mathbb{T} .

Hence, T_l must be committed in \mathbb{T} . Then since $T_l \dot{<}^x_{\mathbb{T}} T_i$ (so $\hat{T}_l <_{\hat{S}_H} \hat{T}_i$), $op_l \in \text{Vis}(\hat{S}_H, T_i)$.

Since e'_u is non-local, then it is not followed in $\mathbb{T}|T_l$ by another $e''_u = \circ_{s_l}(x)v$. Since $op_l \rightsquigarrow e'_u$, $\#op'_l = w_l(x)v' \rightarrow ok_l$ s.t. $op_l <_{\mathbb{T}|T_i} op'_l$.

Let T_m be a transaction in \mathbb{T} s.t. $T_l \dot{<}^x_{\mathbb{T}} T_m \dot{<}^x_{\mathbb{T}} T_j$. If $T_m \in \psi_{\mathbb{T}}(T_i, x)$, then from Lemma 38, T_m is not committed in \mathbb{T} . If $T_m \notin \psi_{\mathbb{T}}(T_i, x)$, then from Lemma 39, T_m is also not committed in \mathbb{T} . In either case, \hat{T}_m is aborted in \hat{S}_H . Thus, for any T_m s.t. $T_l \dot{<}^x_{\mathbb{T}} T_m \dot{<}^x_{\mathbb{T}} T_j$, $\hat{S}_H|\hat{T}_m \not\subseteq \text{Vis}(\hat{S}_H, i)$. Therefore, for any write operation execution $op_m = w_m(x)v'' \rightarrow ok_m \in \mathbb{T}|T_m$, $op_m \notin \text{Vis}(\hat{S}_H, T_i)$.

Let $T_m \in \mathbb{T}$ be any transaction s.t. $T_j \dot{<}^x_{\mathbb{T}} T_m \dot{<}^x_{\mathbb{T}} T_i$. Since e_a consonant, it is needed, so $\exists e''_u = s_j(x)v'' \in \mathbb{T}|T_j$ s.t. $e''_u <_{\mathbb{T}} e_a$. In addition, since $T_j \dot{<}^x_{\mathbb{T}} T_m \dot{<}^x_{\mathbb{T}} T_i$, then by definition of isolation order, $\exists e = s_m(x)v''' \in \mathbb{T}|T_m$, or $e = g_m(x)v''' \in \mathbb{T}|T_m$, so $e''_u <_{\mathbb{T}} e <_{\mathbb{T}} e_v$. Since $e_a <_{\mathbb{T}} e_v$, then $e''_u <_{\mathbb{T}} e <_{\mathbb{T}} e_u$. Thus, by definition of abort accord, T_m is not committed in \mathbb{T} , so, by construction, \hat{T}_m is aborted in \hat{S}_H . Thus, for any T_m s.t.

$T_j \dot{<}^x_{\mathbb{T}} T_m \dot{<}^x_{\mathbb{T}} T_i$, $\hat{S}_H | \hat{T}_m \not\subseteq \text{Vis}(\hat{S}_H, i)$. Therefore, for any write operation execution $op_m = w_m(x)v'' \rightarrow ok_m \in \mathbb{T} | T_m$, $op_m \notin \text{Vis}(\hat{S}_H, T_i)$.

Since no other write operation execution follows op_l in $\mathbb{T} | T_m$, and since there is no transaction $T_m \in \mathbb{T}$ s.t. $\hat{T}_l <_{\hat{S}_H} \hat{T}_m <_{\hat{S}_H} \hat{T}_i$ (and therefore $\hat{T}_l <_{\text{Vis}(\hat{S}_H, \hat{T}_i)} \hat{T}_m <_{\text{Vis}(\hat{S}_H, \hat{T}_i)} \hat{T}_i$) s.t. $\exists op_m = w_m(x)v'' \rightarrow ok_m \in \mathbb{T} | T_m$ and $op_m \in \text{Vis}(\hat{S}_H, \hat{T}_i)$, then $w_l(x)v \rightarrow ok_l <_{\text{Vis}(\hat{S}_H, \hat{T}_i)} op_i$ and $v \neq 0$.

□

Corollary 6 (Total Non-local Read Consistency). *By extension of the above, since, by definition, if for some sequential history S , $\hat{S}_H | tr_j \in \text{Vis}(S, T_i)$, then $\text{Vis}(\hat{S}_H, T_j)$ is a prefix of $\text{Vis}(S, T_i)$, then for any $op_j \in \mathbb{T} | T_j$ s.t. $op_j = r_i(op_j) \rightarrow v$ either:*

- 1) $v \neq 0$ and $\exists op_k \in \text{Vis}(\hat{S}_H, T_k)$ for some T_k s.t. $op_k = w_k(x)v \rightarrow ok_j$, $op_k <_{\text{Vis}(\hat{S}_H, \hat{T}_i)} op_j$, or
- 2) $v = 0$ and $\nexists op_k \in \text{Vis}(\hat{S}_H, T_i)$ s.t. $op_k = w_k(x)v \rightarrow ok_k$ and $op_k <_{\text{Vis}(\hat{S}_H, T_i)} op_j$.

Lemma 47 (Local Read Consistency). *For any $op_i \in \mathbb{T} | T_i$ s.t. $op_i = r_i(op_i) \rightarrow v$ and op_i is local, then $\exists op'_i \in \text{Vis}(\hat{S}_H, T_i)$ s.t. $op'_i = w_i(x)v \rightarrow ok_i$, $op'_i <_{\text{Vis}(\hat{S}_H, T_i)} op_i$.*

Proof. From local read consonance it follows that For any $op_i \in \mathbb{T} | T_i$ s.t. op_i is local, then $\exists op'_i \in \mathbb{T} | T_i$ and $op'_i <_{\mathbb{T} | T_i} op_i$. Thus, $op'_i <_{\hat{S}_H | \hat{T}_i} op_i$. Since $T_i \subseteq \text{Vis}(\hat{S}_H, T_i)$ then $op'_i <_{\text{Vis}(\hat{S}_H, T_i)} op_i$. □

Corollary 7 (Total Local Read Consistency). *For any $op_i \in \mathbb{T} | T_i$ s.t. $op_i = r_i(op_i) \rightarrow v$ and op_i is local, then for any T_j , $\exists op'_i \in \text{Vis}(\hat{S}_H, T_j)$ s.t. $op'_i = w_i(x)v \rightarrow ok_i$, $op'_i <_{\text{Vis}(\hat{S}_H, T_j)} op_i$.*

Let there instead be such $T_i \in \mathbb{T}$ that T_i is either committed or not committed \mathbb{T} . Then:

Lemma 48 (Non-local Read Last-use Consistency). *For any $op_i \in \mathbb{T} | T_i$ s.t. $op_i = r_i(op_i) \rightarrow v$ and op_i is non-local, then either*

- i) $v \neq 0$ and $\exists op_j \in L\text{Vis}(\hat{S}_H, \hat{T}_i)$ for some T_j s.t. $op_j = w_j(x)v \rightarrow ok_j$, $op_j <_{L\text{Vis}(\hat{S}_H, \hat{T}_i)} op_i$, or
- ii) $v = 0$ and $\nexists op_j \in L\text{Vis}(\hat{S}_H, \hat{T}_i)$ s.t. $op_j = w_j(x)v \rightarrow ok_j$ and $op_j <_{L\text{Vis}(\hat{S}_H, \hat{T}_i)} op_i$.

Proof. Since op_i is consonant and non-local, then $\exists e_v = g_i(x)v \in \mathbb{T}$, s.t. $op_i \leftarrow e_v$. Then, since \mathbb{T} is consonant, then e_v is also consonant, so one of the following is true:

- a) $v = 0$ and $\nexists e_u = s_j(x) \square \in \mathbb{T}$ for some $T_j \in \mathbb{T}$ s.t. $e_u <_{\mathbb{T}} e_v$.

In which case, if $\nexists op_j = w_j(x) \square \rightarrow ok_i \in \mathbb{T} | T_j$ s.t. $op_j <_{\mathbb{T}} g_i(x)v$, then, $\nexists T_j \in \mathbb{T}$ s.t. $T_j \dot{<}_{\mathbb{T}} T_i$ and $op_j \in \mathbb{T} | T_j$. Thus, from construction of \hat{S}_H ,

$\nexists \hat{T}_j \in \mathbb{T}$ s.t. $\hat{T}_j <_{\hat{S}_H} \hat{T}_i$ and $op_j \in \hat{S}_H|\hat{T}_j$. Thus, from construction of $LVis(\hat{S}_H, \hat{T}_i)$, for any such T_j , $\hat{S}_H|\hat{T}_j \not\subseteq LVis(\hat{S}_H, \hat{T}_i)$, so for any such T_j , $w_j(x)v \rightarrow ok_j \notin LVis(\hat{S}_H, \hat{T}_i)$ and $v = 0$.

On the other hand, if $\exists op_j w_j(x) \square \rightarrow ok_i \in \mathbb{T}|T_j$ s.t. $op_j <_{\mathbb{T}} g_i(x)v$, then if T_j is committed in \mathbb{T} , then, from the definition of commit write obligato, $\exists \circ s_j(x) \square \in \mathbb{T}|T_j$, which contradicts the assumption of case a). If, however, T_j is not committed in \mathbb{T} , then either T_j is decided on x in \mathbb{T} , or it is not. In the former of those two cases, from the definition of closing write obligato, $\exists \circ s_j(x) \square \in \mathbb{T}|T_j$, which also contradicts the assumption of case a). In the latter case, since \hat{T}_j is neither committed in \mathbb{T} nor decided on x in \mathbb{T} , then neither is it committed in \hat{S}_H nor decided on x in \hat{S}_H . Therefore, by definition of $LVis(\hat{S}_H, \hat{T}_i)$, $\hat{S}_H|\hat{T}_j|x \not\subseteq LVis(\hat{S}_H, \hat{T}_i)$. Thus for any such T_j , $w_j(x) \square \rightarrow ok_j \notin LVis(\hat{S}_H, \hat{T}_i)$ and $v = 0$.

b) $v \neq 0$ and $\exists e_u = \circ s_j(x)v \in \mathbb{T}$ for some $T_j \in \mathbb{T}$ s.t. $e_u <_{\mathbb{T}} e_v$.

Since e_u is consonant, then $\exists op_j = w_j(x)v \rightarrow ok_j \in \mathbb{T}|T_j$ s.t. op_j is non-local and consonant, and $op_j < \circ s_j(x)v$. Thus, since $e_u <_{\mathbb{T}} e_v$, $T_j <_{\mathbb{T}}^x T_i$, then, by construction, $\hat{T}_j <_{\hat{S}_H} \hat{T}_i$.

If T_i is not committed in \mathbb{T} , since $T_i \dot{\sim} T_j$, and since \mathbb{T} is decisive, then T_j is decided on x in \mathbb{T} . Thus, from Def. 32, $\hat{S}_H|\hat{T}_j|x \subseteq LVis(\hat{S}_H, T_i)$, and therefore $op_j <_{LVis(\hat{S}_H, T_i)} op_i$. Alternatively, if T_i is committed in \mathbb{T} , then, from commit accord, T_j is also committed in \mathbb{T} . Then, by definition of $LVis(\hat{S}_H, T_i)$, $\hat{S}_H|\hat{T}_j \subseteq LVis(\hat{S}_H, T_i)$, and thus $op_j <_{LVis(\hat{S}_H, T_i)} op_i$. Then, from Lemma 36, $op_j <_{LVis(\hat{S}_H, T_i)} op_i$. Thus, $w_j(x)v \rightarrow ok_j <_{LVis(\hat{S}_H, \hat{T}_i)} op_i$ and $v \neq 0$.

c) $\exists e_a^j = \diamond s_j(x)v \in \mathbb{T}|T_j$ s.t. $e_a^j <_{\mathbb{T}} e_v$.

Since \mathbb{T} is consonant, then e_a^j is consonant, so e_a^j is conservative, and thus $\exists e_v^j = g_j(x)v \in \mathbb{T}|T_j$.

From Corollary 5 $\forall T_n \in \psi_{\mathbb{T}}(T_i, x)$ ($n \neq i$) $T_n <_{\mathbb{T}}^x T_i$. So, by construction, for every such T_n , \hat{T}_n is aborted in \hat{S}_H and therefore not included as a whole in $LVis(\hat{S}_H, T_i)$. In addition, since e_a^n is needed there is a preceding routine update $e_u^n = \circ s_n(x)v'$. Because of unique writes, $v' \neq v$. Therefore, it is not true that $T_i \dot{\sim} T_n$. Furthermore, since $e_u^n <_{\mathbb{T}} e_a^n$ and $e_a^n <_{\mathbb{T}} e_a^j$ and $e_a^j <_{\mathbb{T}} e_v$, then, $e_u^n <_{\mathbb{T}} e_a^n <_{\mathbb{T}} e_v$, so, from chain isolation, $\nexists \xi(\mathbb{T}, T_n, T_m)$. Thus, from Def. 32 $\hat{S}_H|\hat{T}_n$ is not included in $LVis(\hat{S}_H, T_i)$. Thus for any $T_n \in \psi_{\mathbb{T}}(T_i, x)$ ($n \neq i$), $\forall op_n = w_n(x)v \rightarrow \in \mathbb{T}|T_n$, $op_n \notin LVis(\hat{S}_H, T_i)$.

Given $\psi_{\mathbb{T}}(T_i, x)$, from Lemma 37, $\exists e_v^k = g_k(x)v \in \mathbb{T}|T_k$ s.t. T_k is the first element of $\psi_{\mathbb{T}}(T_i, x)$ that is initial and non-local, and either of the following is true:

i) $v = 0$ and $\nexists T_l \in \mathbb{T}$ s.t. $e_u^l = s_l(x) \square \in \mathbb{T}|T_l$ and $e_u^l <_{\mathbb{T}} e_v^k$.

Then, either $\exists e_u^l = s_l(x)v \in \mathbb{T}|T_l$ and $e_v^k <_{\mathbb{T}} e_u^l$ or $\nexists e_u^l = s_l(x) \square \in \mathbb{T}|T_l$.

If $\exists e_u^l = s_l(x) \square \in \mathbb{T}|T_l$ and $e_v^k <_{\mathbb{T}} e_u^l$, then from Lemma 37, $e_u^l = \circ s_l(x) \square \in \mathbb{T}|T_l$. Thus, by definition of isolation order, $T_k <_{\mathbb{T}}^x T_l$.

Thus, if $T_l <_{\mathbb{T}}^x T_i$, then, from Lemma 39, T_l is aborted or live in \mathbb{T} , so, by construction, \hat{T}_l is aborted in \hat{S}_H . Since \hat{T}_l is not committed in \hat{S}_H , then

$\hat{T}_l | \hat{S}_H$ is not included as a whole in $LVis(\hat{S}_H, T_i)$. Furthermore, $\hat{S}_H | \hat{T}_l$ can be omitted from $LVis(\hat{S}_H, T_i)$. From unique routine updates, there cannot be $T_n \in \xi(\mathbb{T}, T_l, T_i)$ s.t. $\exists \circ s_n(x)0$, so since $v = 0$ and from self-containment, $\nexists \xi(\mathbb{T}, T_l, T_i)$. Thus, from Def. 32, $\hat{S}_H | \hat{T}_l \not\subseteq LVis(\hat{S}_H, T_i)$. Then, for any write operation execution $op_l = w_l(x)v' \rightarrow ok_l$ in any such T_l , $op_l \notin LVis(\hat{S}_H, T_i)$ (and $v = 0$).

Alternatively, if $T_i \prec_{\mathbb{T}}^x T_l$, then since $e_a \prec_{\mathbb{T}} e_v$, then $e_v \prec_{\mathbb{T}} e_u^l$, so $T_i \prec_{\mathbb{T}}^x T_l$, and thus $T_i \prec_{\hat{S}_H} T_l$, which means that $\hat{S}_H | \hat{T}_l \not\subseteq LVis(\hat{S}_H, T_i)$. In either case for any write operation execution $op_l = w_l(x)v' \rightarrow ok_l$ in any such T_l , $op_l \notin Vis(\hat{S}_H, T_i)$ (and $v = 0$).

On the other hand, if $\nexists e_u^l = s_l(x) \square \in \mathbb{T} | T_l$, either $T_l \prec_{\mathbb{T}} T_i$, or $T_l \dot{\prec}_{\mathbb{T}} T_i$. In the latter case, if $T_i \prec_{\mathbb{T}} T_l$, then, trivially, no subset of $\hat{S}_H | \hat{T}_l$ is contained in $LVis(\hat{S}_H, T_i)$. If $T_i \dot{\prec}_{\mathbb{T}} T_l$, then there does not exist $\xi(\mathbb{T}, T_l, T_i)$, so, from Def. 32, no subset of $\hat{S}_H | \hat{T}_l$ is contained in $LVis(\hat{S}_H, T_i)$. If $T_l \prec_{\mathbb{T}} T_i$, then either $\mathbb{T} | T_l$ contains some write operation $op_l = w_l(x)v' \rightarrow ok_l$ or it does not. However, from view write obligato, since $T_l \prec_{\mathbb{T}} T_i$, there must be $s_l(x) \square \in \mathbb{T} | T_l$, so, there is no such op_l . Then vacuously, for any write operation execution $op_l = w_l(x) \square \rightarrow ok_l$ in any such T_l , $op_l \notin LVis(\hat{S}_H, T_i)$ (and $v = 0$).

- ii) $v \neq 0$ and $\exists T_l \in \mathbb{T}$ s.t. $e_u^l = \circ s_l(x)v \in \mathbb{T} | T_l$ and $e_u^l \prec_{\mathbb{T}} e_v'$.

Since \mathbb{T} is consonant, then e_u^l is consonant, so $\exists op_l = w_l(x)v \rightarrow ok_l$ s.t. $op_l \rightsquigarrow e_u^l$.

Let us first assume that T_l is committed in \mathbb{T} . Then since $T_l \prec_{\mathbb{T}}^x T_i$ (so $\hat{T}_l \prec_{\hat{S}_H} \hat{T}_i$), $op_l \in LVis(\hat{S}_H, T_i)$.

If, on the other hand, T_l is not committed in \mathbb{T} , then, since e_v^k is consonant, then e_u^l is the ultimate routine update event in $\mathbb{T} | T_l$. Therefore, from decisiveness, e_u^l is either the closing routine update event on x in $\mathbb{T} | T_l$, or $e_u^l \prec_{\mathbb{T}} res_l(C_l) \prec_{\mathbb{T}} e_v$. Since T_l is not committed in \hat{S}_H , e_u^l is the closing routine update event, so op_l is the closing write on x in T_l . Because of this, and since $\hat{T}_l \prec_{\hat{S}_H} \hat{T}_i$, $\hat{S}_H | \hat{T}_l$ can be included in $LVis(\hat{S}_H, T_i)$. Then, since $T_i \rightsquigarrow T_l$, then there exists $\xi(\mathbb{T}, T_l, T_i)$, so according to Def. 32, $\hat{S}_H | \hat{T}_l$ is included in $LVis(\hat{S}_H, T_i)$, and therefore $op_l \in LVis(\hat{S}_H, T_i)$.

From minimalism, e_u^l is not followed in $\mathbb{T} | T_l$ by another $\circ s_l(x) \square$. Since $op_l \rightsquigarrow e_u^l$, $\nexists op_l' = w_l(x) \square \rightarrow ok_l$ s.t. $op_l \prec_{\mathbb{T} | T_l} op_l'$.

Let T_m be any transaction in \mathbb{T} s.t. $T_l \prec_{\mathbb{T}} T_m \prec_{\mathbb{T}} T_i$. If $\nexists w_m(x) \square \rightarrow ok_m \in \mathbb{T} | T_m$, then trivially, for any such T_m $\nexists w_m(x)v \rightarrow ok_m \in LVis(\hat{S}_H, T_i)$. Thus, let there be $op_m = w_m(x)v' \rightarrow ok_m \in \mathbb{T} | T_m$.

If $T_m \in \psi_{\mathbb{T}}(T_{\mathbb{T}}, x)i$, then, as shown above, $\hat{S}_H | \hat{T}_m \not\subseteq LVis(\hat{S}_H, T_i)$ and therefore $op_m \notin LVis(\hat{S}_H, T_i)$. Hence, let $T_m \notin \psi_{\mathbb{T}}(T_{\mathbb{T}}, x)i$.

Either $tr_m \prec_{\mathbb{T}} T_i$ or $tr_m \dot{\prec}_{\mathbb{T}} T_i$. In the latter case, if T_m is not committed, T_m can be excluded from $LVis(\hat{S}_H, T_i)$. Since in that case there does not exist $\xi(\mathbb{T}, T_m, T_i)$, then, by Def. 32, $\hat{S}_H | \hat{T}_m \not\subseteq LVis(\hat{S}_H, T_i)$ and therefore $op_m \notin LVis(\hat{S}_H, T_i)$. If T_m is committed, then if $w_m(x)v' \rightarrow ok_m \in \mathbb{T} | T_m$, then by commit write obligato, there would have to exist $e_u^m = \circ s_m(x)v' \in \mathbb{T} | T_m$, which would imply that $T_m \prec_{\mathbb{T}}^x T_i$, which

contradicts the assumption that $tr_m \dot{\prec}_{\mathbb{T}} T_i$. Therefore, there is no such transaction.

If $T_m \dot{\prec}_{\mathbb{T}} T_i$, then, if $T_m \dot{\prec}_{\mathbb{T}} T_l$, then $\hat{T}_m \prec_{\hat{S}_H} \hat{T}_l$, and therefore $op_m \prec_{LVis(\hat{S}_H, T_i)} op_l$, which has no bearing on whether op_i is preceded by a corresponding write operation. Hence, let $T_l \dot{\prec}_{\mathbb{T}} T_m \dot{\prec}_{\mathbb{T}} T_i$. Then, T_m is either committed in \mathbb{T} or not.

If T_m is committed, then from commit write obligato $\exists e_u^m = \circ s_m(x)v' \in \mathbb{T}|T_m$. From isolation, it is impossible that $T_i \dot{\prec}_{\mathbb{T}} T_m$ or $T_m \dot{\prec}_{\mathbb{T}} T_l$, then $T_l \dot{\prec}_{\mathbb{T}} T_m \dot{\prec}_{\mathbb{T}} T_i$.

Then, if $T_j \dot{\prec}_{\mathbb{T}} T_m \dot{\prec}_{\mathbb{T}} T_i$, since e_a is consonant, it is needed, so $\exists e_u^j = s_j(x) \square \in \mathbb{T}|T_j$ s.t. $e_u^j \prec_{\mathbb{T}} e_a$. In addition, since $T_j \dot{\prec}_{\mathbb{T}} T_m \dot{\prec}_{\mathbb{T}} T_i$, then by definition of isolation, $\exists e = s_m(x) \square \in \mathbb{T}|T_m$, or $eg_m(x) \square \in \mathbb{T}|T_m$, so $e_u^j \prec_{\mathbb{T}} e \prec_{\mathbb{T}} e_v$. Since $e_a \prec_{\mathbb{T}} e_v$, then $e_u^j \prec_{\mathbb{T}} e \prec_{\mathbb{T}} e_u$. Thus, by definition of abort accord, T_m cannot be in \mathbb{T} , thus there is no such T_m .

If $T_l \dot{\prec}_{\mathbb{T}} T_m \dot{\prec}_{\mathbb{T}} T_j$, then, since $T_m \notin \psi_{\mathbb{T}}(T_{\mathbb{T}}, x)i$, then, from Lemma 39, T_m cannot be committed in \mathbb{T} , thus, there is also no such T_m .

Since there is no such T_m that is both committed and contains an operation execution such as op_m , then for any such T_m , $\hat{S}_H|\hat{T}_m \not\subseteq LVis(\hat{S}_H, T_i)$ and therefore $op_m \notin LVis(\hat{S}_H, T_i)$.

On the other hand, if T_m is not committed in \mathbb{T} , then, since op_m is consonant, either $op_m \rightsquigarrow e_u^m$ where $e_u^m = \circ s_m(x)v' \in \mathbb{T}|T_m$, or $\sharp \circ s_m(x) \square \in \mathbb{T}|T_m$. Since, from view write obligato the latter case is impossible, then $\exists e_u^m = \circ s_m(x)v' \in \mathbb{T}|T_m$. Then, from the definition of isolation order, $T_l \dot{\prec}_{\mathbb{T}} T_m \dot{\prec}_{\mathbb{T}} T_i$. Since T_m is not committed, then \hat{T}_m can be omitted in $LVis(\hat{S}_H, T_i)$. Due to unique routine updates, there cannot be any $T_n \in \mathbb{T}$ s.t. $\circ s_n(x)v''$ where $v'' = v$. Therefore, given any $\xi(\mathbb{T}, T_m, T_i)$ and there is no transaction to satisfy self-containment. Thus, there is no such $\xi(\mathbb{T}, T_m, T_i)$. Thus, by Def. 32, $\hat{S}_H|\hat{T}_m \not\subseteq LVis(\hat{S}_H, T_i)$ and therefore $op_m \notin LVis(\hat{S}_H, T_i)$.

Because there is no T_m s.t. $\hat{S}_H|\hat{T}_m \subseteq LVis(\hat{S}_H, T_i)$ and $op_m \in LVis(\hat{S}_H, T_i)$, then there is no write operation execution op' on x s.t. $op_l \prec_{LVis(\hat{S}_H, T_i)} op_i$. Therefore $w_l(x)v \rightarrow ok_l \prec_{Vis(\hat{S}_H, \hat{T}_i)} op_i$ and $v \neq 0$.

□

Lemma 49 (All Non-local Read Last-use Consistency). *If for some sequential history S , $\hat{S}_H|tr_j \in Vis(S, T_i)$, for any $op_j \in \mathbb{T}|T_j$ s.t. $op_j = r_i(op_j) \rightarrow v$ either:*

- 1) $v \neq 0$ and $\exists op_k \in Vis(\hat{S}_H, \hat{T}_k)$ for some T_k s.t. $op_k = w_k(x)v \rightarrow ok_j$, $op_k \prec_{Vis(\hat{S}_H, \hat{T}_i)} op_j$, or
- 2) $v = 0$ and $\sharp op_k \in Vis(\hat{S}_H, \hat{T}_i)$ s.t. $op_k = w_k(x)v' \rightarrow ok_k$ and $op_k \prec_{Vis(\hat{S}_H, \hat{T}_i)} op_j$.

Proof. Either $v \neq 0$ or $v = 0$.

- 1) If $v \neq 0$, from Lemma 48, since $v \neq 0$ then $\exists op_k = w_k(x)v \rightarrow ok_k \in \mathbb{T}|T_k$ s.t. $op_k \prec_{LVis(\hat{S}_H, T_j)} op_j$.

From Lemma 41, $\forall T_l$ if $\hat{T}_l \stackrel{\diamond}{\subseteq} LVis(\hat{S}_H, T_j)$ then $\hat{T}_l \stackrel{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$ and $LVis(\hat{S}_H, T_i)|T_l = LVis(\hat{S}_H, T_j)|T_l$. Hence, $T_k \stackrel{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$ and $op_k \in$

$LVis(\hat{S}_H, T_i)$. Furthermore, if $T_k \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_j)$, then $T_k <_{\hat{S}_H} T_j$, so $op_k <_{LVis(\hat{S}_H, T_i)} op_j$.

For the sake of contradiction, let us assume there exists $op_l = w_l(x)v \rightarrow ok_l \in \mathbb{T}|T_l$ s.t. $op_k <_{LVis(\hat{S}_H, T_i)} op_l <_{LVis(\hat{S}_H, T_i)} op_j$. Since from Lemma 48, there is no such transaction in $LVis(\hat{S}_H, T_i)$, then T_l is such that $\hat{T}_l \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_i)$ s.t. and $\hat{T}_l \not\subseteq LVis(\hat{S}_H, T_j)$ (and $\hat{T}_l <_{\hat{S}_H} \hat{T}_j$).

If $\hat{T}_l \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_i)$, then, by definition, T_l is not committed in \mathbb{T} .

If $T_l \dot{<}_x \mathbb{T}T_j$, then this is a contradiction by analogy to Lemma 48.

If $T_j \dot{<}_x \mathbb{T}T_l$, then, from Def. 31, $\hat{T}_j <_{\hat{S}_H} \hat{T}_l$, which implies that $op_j <_{LVis(\hat{S}_H, T_j)} op_l$, which is a contradiction.

If $T_l \dot{<}_x \mathbb{T}T_j$, then $\#e_u^k = \circ s_l(x) \square \in \mathbb{T}|T_l$. Hence, from Def. 31, $\hat{T}_j <_{\hat{S}_H} \hat{T}_l$, which implies that $op_j <_{LVis(\hat{S}_H, T_j)} op_l$, which, again, is a contradiction.

Thus, there is no such T_l , and, therefore, $op_k <_{LVis(\hat{S}_H, T_i)} op_j$ (and $v \neq 0$).

- 2) If $v = 0$, let us assume by contradiction, that there exists such T_k and op_k . From Lemma 48, since $v = 0$ then $\#op_l = w_l(x) \square \rightarrow ok_l \in \mathbb{T}|T_l$ s.t. $op_l <_{LVis(\hat{S}_H, T_j)} op_j$. Hence, T_k must be such that $\hat{T}_k \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_i)$ s.t. and $\hat{T}_k \not\subseteq LVis(\hat{S}_H, T_j)$ (and $T_k <_{\hat{S}_H} T_j$).

From Lemma 42, $\forall T_l$ if $\hat{T}_l \not\subseteq LVis(\hat{S}_H, T_j)$ and $\exists \xi(\mathbb{T}, T_l, T_j)$ then $\hat{T}_l \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_i)$. Thus, if T_k is such that $\hat{T}_k \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_j)$ and $\hat{T}_k \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_i)$, then $\# \xi(\mathbb{T}, T_k, T_j)$.

If $\hat{T}_k \stackrel{\circ}{\subseteq} LVis(\hat{S}_H, T_i)$, then, by definition, T_k is not committed in \mathbb{T} .

If $T_k \dot{<}_x \mathbb{T}T_j$, then this is a contradiction by analogy to Lemma 48.

If $T_j \dot{<}_x \mathbb{T}T_k$, then, from Def. 31, $\hat{T}_j <_{\hat{S}_H} \hat{T}_k$, which implies that $op_j <_{LVis(\hat{S}_H, T_j)} op_k$, which is a contradiction.

If $T_k \dot{<}_x \mathbb{T}T_j$, then $\#e_u^k = \circ s_k(x) \square \in \mathbb{T}|T_k$. Hence, from Def. 31, $\hat{T}_j <_{\hat{S}_H} \hat{T}_k$, which implies that $op_j <_{LVis(\hat{S}_H, T_j)} op_k$, which, again, is a contradiction.

Thus, there is no such T_k , and, therefore, $\#op_k \in LVis(\hat{S}_H, \hat{T}_i)$ s.t. $op_k = w_k(x)v' \rightarrow ok_k$ and $op_k <_{LVis(\hat{S}_H, \hat{T}_i)} op_j$ (and $v = 0$).

□

Lemma 50 (Local Read Last-use Consistency). *For any $op_i \in \mathbb{T}|T_i$ s.t. $op_i = r_i(op_i) \rightarrow v$ and op_i is local, then $\exists op'_i \in LVis(\hat{S}_H, T_i)$ s.t. $op'_i = w_i(x)v \rightarrow ok_i$, $op'_i <_{LVis(\hat{S}_H, T_i)} op_i$.*

Proof. From local read consonance it follows that for any $op_i \in \mathbb{T}|T_i$ s.t. op_i is local, then $\exists op'_i \in \mathbb{T}|T_i$ and $op'_i <_{\mathbb{T}|T_i} op_i$. Thus, $op'_i <_{\hat{S}_H|\hat{T}_i} op_i$. Since op_i and op_j operate on the same variable, then trivially, $op_i \in LVis(\hat{S}_H, T_i) \iff op'_i \in LVis(\hat{S}_H, T_i)$. Hence, $op'_i <_{Vis(\hat{S}_H, T_i)} op_i$. □

Corollary 8 (Total Local Read Last-use Consistency). *For any $op_i \in \mathbb{T}|T_i$ s.t. $op_i = r_i(op_i) \rightarrow v$ and op_i is local, then for any T_j , $\exists op'_i \in LVis(\hat{S}_H, T_j)$ s.t. $op'_i = w_i(x)v \rightarrow ok_i$, $op'_i \leq_{LVis(\hat{S}_H, T_j)} op_i$.*

Lemma 51 (Total Write Consistency). *For any T_i, T_j , $\forall op_i \in LVis(\hat{S}_H, T_j)$ s.t. $op_i = w_i(op_i)v \rightarrow ok_i \in \mathbb{T}|T_i$, v is in the domain of x .*

Proof. Follows from write consonance. \square

A.5 Proof for Theorem 1

Proof for Theorem 1: Trace Last-use Opacity. For every transaction $T_i \in \mathbb{T}$, given \hat{S}_H constructed by Def. 31,

- i) If T_i is committed in H , from Corollary 6, $\forall op_j \in H|T_j$ s.t. $op_j = r_i(op_j) \rightarrow v$ and op_j is non-local, either:
 - a) $v \neq 0$ and $\exists op_k \in Vis(\hat{S}_H, \hat{T}_k)$ for some T_k s.t. $op_k = w_k(x)v \rightarrow ok_j$, $op_k \leq_{Vis(\hat{S}_H, \hat{T}_i)} op_j$, or
 - b) $v = 0$ and $\nexists op_k \in Vis(\hat{S}_H, \hat{T}_i)$ s.t. $op_k = w_k(x)v \rightarrow ok_k$ and $op_k <_{Vis(\hat{S}_H, \hat{T}_i)} op_j$.

In addition, from Corollary 7, $\forall op_j \in H|T_j$ s.t. $op_j = r_i(op_j) \rightarrow v$ and op_j is local, $\exists op'_j \in Vis(\hat{S}_H, \hat{T}_j)$ s.t. $op'_j = w_j(x)v \rightarrow ok_j$, $op'_j \leq_{Vis(\hat{S}_H, \hat{T}_i)} op_j$. Furthermore, from Lemma 51, $\forall op'_j \in H|T_j$ s.t. $op_j = w_i(op_j)v \rightarrow ok_j$, v is in the domain of x .

Thus, $Vis(\hat{S}_H, T_i)$ is legal, and therefore T_i is legal.

- ii) If T_i is not committed in H , from Lemma 49, $\forall op_j \in H|T_j$ s.t. $op_j = r_i(op_j) \rightarrow v$ and op_j is non-local, either:
 - a) $v \neq 0$ and $\exists op_k \in LVis(\hat{S}_H, \hat{T}_k)$ for some T_k s.t. $op_k = w_k(x)v \rightarrow ok_j$, $op_k \leq_{LVis(\hat{S}_H, \hat{T}_i)} op_j$, or
 - b) $v = 0$ and $\nexists op_k \in LVis(\hat{S}_H, \hat{T}_i)$ s.t. $op_k = w_k(x)v \rightarrow ok_k$ and $op_k <_{LVis(\hat{S}_H, \hat{T}_i)} op_j$.

In addition, from Corollary 8, $\forall op_j \in H|T_j$ s.t. $op_j = r_i(op_j) \rightarrow v$ and op_j is local, $\exists op'_j \in LVis(\hat{S}_H, \hat{T}_j)$ s.t. $op'_j = w_j(x)v \rightarrow ok_j$, $op'_j \leq_{LVis(\hat{S}_H, \hat{T}_i)} op_j$. Furthermore, from Lemma 51, $\forall op'_j \in H|T_j$ s.t. $op_j = w_i(op_j)v \rightarrow ok_j$, v is in the domain of x .

Thus, $LVis(\hat{S}_H, T_i)$ is legal, and therefore T_i is last-use legal.

Since every committed transaction $T_i \in H$ is legal if it is committed and last-use legal if it is not committed, then H is final-state last-use opaque.

Since a prefix of a harmonious \mathbb{T} is trivially also harmonious, then for every prefix \mathbb{T}' of \mathbb{T} , $H' = hist(\mathbb{T})'$ is also final-state last-use opaque. Thus, H is last-use opaque. \square